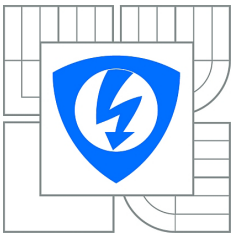
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
ÚSTAV AUTOMATIZACE A MĚŘICÍ TECHNIKY

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION
DEPARTMENT OF CONTROL AND INSTRUMENTATION

ANALÝZA CHOVÁNÍ REALTIME LINUX OS PRO MODULY ARM

REAL TIME LINUX LATENCY ANALYSIS

BAKALÁŘKA PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

PETER KRAJÍČEK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. SOBĚSLAV VALACH

BRNO 2012

Vlozit list zadáni!

Abstrakt
Práce rozebírá příčiny vzniku zpoždění při použití operačního systému GNU/Linux. Popisuje několik zvolených typů analýzy, které pak implementujeme pomocí programovacího jazyka C.
Samotné měření je realizováno na kitu s mikroprocesorem OMAP3530 a jádrem Linux.

Summary
The thesis analyzes the causes of delays when using the operating system GNU/Linux. It describes a few selected types of analysis, which then implemented using the C programming language.
Itself measurement is made on the kit with OMAP3530 microprocessor and the Linux kernel.

Klíčová slova
zpoždění, realtime, Linux, kernel, ARM, BeagleBoard

Keywords
latency, realtime, Linux, kernel, ARM, BeagleBoard

KRAJÍČEK, P. *Analýza chování realtime Linux OS pro moduly ARM*. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, 2012. 61 s. Vedoucí Ing. Soběslav Valach.

Prohlášení

Prohlašuji, že svou bakalářskou práci na téma „Analýza chování realtime Linux OS pro moduly ARM“ jsem vypracoval samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce. Jako autor uvedené bakalářské práce dále prohlašuji, že v souvislosti s vytvořením této bakalářské práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a jsem si plně vědom následků porušení ustanovení S 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení S 152 trestního zákona č. 140/1961 Sb.

V Brně .....

.....
(podpis autora)

Poděkování
Tímto bych chtěl poděkovat vedoucímu bakalářské práce panu Ing. Soběslavu Valachovi za zapůjčení kitu BeagleBoard a podnětné připomínky. Dále bych chtěl poděkovat početné linuxové komunitě za vytvoření manuálových stránek http://kernel.org/doc, které sloužily jako zdroj informací při psaní zdrojových textů.

Peter Krajíček

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Z hlediska realtime chování je, v dnešní době vysokých výpočetních výkonů, majoritním zdrojem negativních vlastností právě softwarové vybavení. Ať už se jedná o samotnou aplikaci, nebo operační systém. V samotné práci se zabýváme analýzou operačního systému Linux a jeho uplatněním pro realtime aplikace. Rozebereme si pojem zpoždění, který přímo ovlivňuje realtime vlastnosti. Dále navrhneme několik metod pro analýzu zpoždění operačních systému. Vybrané z nich implementujeme a vyhodnotíme získané data.

Pro měření byl vybrán modul BeagleBoard, který je osazen procesorem ARM OMAP3530 s jádrem Cortex A8. Zásadním formálním zdrojem při vypracování práce byla dokumentace k danému procesoru od firmy Texas Instrument. Modul disponuje sériovým rozhraním, které bylo využito pro měření. Důležitou součástí modulu je patice pro SD paměťové karty. Právě SD kartu jsme využili pro uložení samotného operačního systému a aplikací pro testy.

Poslední kapitola obsahuje komplexní vyhodnocení a interpretaci naměřených dat. Závěrem jsou shrnuty dosažené výsledky a navrženy možnosti rozsáhlejšího zpracování dané problematiky.

1. Teoretický úvod

První kapitola si klade za úkol shrnout minimální požadavky pro orientaci v problematice. V dnešní době plné elektroniky a datových přenosů se s realtime požadavkem potkáváme velice často. I přesto dané problematice není věnována širší pozornost, jelikož se jedná o skoro nultý požadavek v každém běžně dostupném případě. Mobilní telefon, počítač či různé elektronické hračky musí obsloužit požadavky v očekávaném časovém horizontu.

Například často používaný přenos zvuku i videa má nároky na stabilní časové zpoždění. Nelibě bychom při telefonním hovoru poslouchali hluchá místa. Ať za použití telefonního přístroje či počítače. I když dnešní výkoná výpočetní technika budí dojem téměř okamžité reakce na cokoliv. Nemusí tomu být tak. Uživatel sice má zdání běhu více aplikací zároveň. Ale jak to často bývá, zdání klame. Ve skutečnosti se musí všechny výpočty uskutečnit na jednom či omezeném počtu procesorů. Dojem kontinuálního zpracování je způsoben rychlým přepínáním mezi běžícími procesy. V případě mnoha požadavků najednou je na místě očekávat soupeření o zdroje. A tím jsou některé procesy nuceny čekat na své vykonání.

S realtime problematikou se nepotkáváme jen v civilní sféře. Další příklady bychom našli v průmyslu. Od robotiky přes regulaci po řízení chemických reakcí, navigaci, obchodování a bankovní transakce, ... Jaký společný prvek spojuje tak širokou paletu činností si vysvětlíme v další kapitole.

1.1. Realtime úloha

Doslovný překlad slova realtime by zněl „reální čas“.Na první pohled se jedná o úkoly zpracované v reálném čase. Nemůžeme očekávat, že výsledky dostáváme okamžitě. Už ze své podstaty neproběhne zpracování digitální informace za nulový čas. Jádro procesoru potřebuje několik taktů pro zpracování vstupu. Tento počet ani nemusí být předem znám. I analogový systém bude mít nějaké časové zpoždění. Dále převody mezi analogovým a digitálním signálem probíhají za nezanedbatelné časové intervaly. Nenulové zpoždění bude obsahovat každý reálný systém.

Podle normy POSIX 1003.1b (realtime extensions) je realtime operační systém definovaný jako: ”Realtime in operating systems: the ability of the operating system to provide a required level of service in a bounded response time.” Tedy schopnost operačního systému poskytnou požadovanou úroveň služeb v omezeném časovém úseku. Definujme realtime dvěma následujícími kritérii.

1.1.1. Časové požadavky na RT

U každé úlohy můžeme stanovit časové zpoždění výsledku, které má zanedbatelný vliv na fungování samotné úlohy a úkonů navazujících na ni. Tento čas nazývejme deadline. Správný výsledek výpočtu obdržený po tomto čase vyhodnocujeme jako chybu. Realtime úloha definuje požadavky na rychlost zpracování. Hodnota deadline je značně variabilní. Její různost se mění růzností aplikace se kterou se spojuje. Kupříkladu řízení asynchronního motoru v elektromobilu bude potřebovat daleko rychlejší reakce, než řízení teploty v bazénu.

Kromě deadline nás bude zajímat i průměrná doba odbavení požadavku. Při telefonickém hovoru nepoznáme rozdíl mezi 10 ms a 20 ms zpoždění. Zato bezpečně poznáme mezeru, která vznikne prudkým skokovým zvýšením zpoždění při přenosu.

1.1.2. Požadavky RT na odolnost vůči chybám

Realtime (RT) úloha se musí vykonat. Tato podmínka klade na systém který úlohu zpracovává nároky na funkčnost za každých podmínek. Tyto podmínky mohou mít fyzikální charakter (pracovní teplota okolí, vlhkost vzduchu, akceptovatelné vibrace či nárazy, ...). Dále funkčnost ohrozí i vstup (výstup) mimo předpokládaný rozsah. Nezapomínejme, že i pozdě ukončený výpočet, tedy překročení deadline, je nechtěná situace. Všechny tyto a další stavy je nutné analyzovat. Vyhodnotit jejich riziko pro funkčnost celku. Systém je musí odchytit a adekvátně na ně reagovat. Jeli systém schopen očekávaně reagovat na všechny anomálie, pak ho nazývejme fault-tolerant systém.

Tady se dostáváme zpátky k bankovním převodům a obchodním transakcím. V dnešní době značná část finančních operací je zpracovávána elektronicky. Je oprávněné očekávat hodnověrnost zobrazených informací a bezpečné uložení, vykonání, zaslaných dat.

1.1.3. Rozdělení realtime systémů

RT systémy můžeme rozdělovat na hard a soft. Z historického hlediska mají hard RT systémy velice nízkou latenci, čili zpoždění. A to v rozsahu od milisekund po mikrosekundy [6]. Nejčastěji se jedná o jednoprocesorové embedded systémy bez podpory více uživatelů, vláken, grafického rozhraní, přímé síťové podpory. Nemívají ani obzvlášť výkonná jádra. Průměrné osobní počítače mívají mnohonásobně vyšší výpočetní výkon. Také neposkytují rozsáhlé možnosti připojení k periferiím. Ve většině případů se nejedná ani o operační systém. Jen běžící úlohu v nekonečném cyklu, která vykonává dané úkoly. Takto čitelné vykonávání se lehce analyzuje ze zdrojových kódů. Matematicky je možné stanovit maximální možnou latenci. V případě operačních systému je velice obtížné exaktně deklarovat latenci. Neexistuje jednoznačný průběh kódu. Kvalifikace latencí nebývá prováděna analýzou, ale statistickým vyhodnocováním naměřených dat. Ostatní systémy podporující RT služby, jenže s vyšší latencí, jsou soft RT systémy. Zde zahrnujeme většinu UNIX-like systému, jelikož mívají v jádru plánovače s podporou plánovaní podle priorit. Je důležité si uvědomit, jak jsme už naznačili, že RT úloha nemusí nutně běžet pod operačním systémem. Operační systém má jisté výhody, ale i nevýhody, které si popíšeme v následující části.

Novější pohled na dělení hard a soft RT systémů je podle případných následků při nedodržení deadline. U hard RT systému je předpoklad, že nedodržení deadline může ohrožovat životy, zdraví či majetek. Opakem je soft RT systém. Kde překročení deadline nemá fatální následky. Může se jednat jen o zhoršení kvality výsledku v daný moment. Příkladem může být rozdíl mezi chybou v kardiostimulátoru a neobdržením paketem v přenosu VoiP. První selhání přímo ohrožuje lidský život. Zatímco druhé nepředstavuje žádné nebezpečí. V nepříliš častém výskytu je mozek člověka schopný doplnit chybějící informaci. Takže výpadek paketu ani nemusí představovat neakceptovatelné snížení kvality.

Dále si můžeme relatime systémy (RTS) rozdělit podle strategie přidělování výpočetního výkonu.

- **Systém běžící v nekonečné smyčce**
  Jedná se o nejjednodušší RTS. Proces běží v nekonečné smyčce, více procesů může běžet v smyčkách za sebou. Nemusí se přepínat mezi procesy. Každý proces se dostává na řadu v periodických intervalech. Použití je například vhodné pro obsluhu vysokorychlostních datových přenosů, nebo sběr dat ze snímačů. Méně vhodné je použití v komplexních systémech. Smyčky spouštěné v krátkých intervalech zbytečně mrhají výkonem CPU. Nepřehlédnutelnou nevýhodou je nastane-li v jedné ze smyček chyba, která způsobí pád běhu programu. Pak se zastaví vykonávání všech smyček, tedy RTS přestane fungovat.

- **RTS řízen přerušením**
  Podle Stallings [17] můžeme přerušení, neboli IRQ, definovat jako událost, která změní posloupnost procesorem vykonávaných instrukcí . Jedná se o hardwarový, nebo softwarový signál upozorňující na nutnost odbavení vzniklého stavu.

- **Více vláknový RTS**
  Na jednom procesoru se zpracovává více procesů. To se může dít pomocí zacyklení více vláken do nekonečné smyčky, přidělováním jednotného časového kvanta každému vláknu, nebo přepínáním mezi vlákny podle definovaných parametrů. Typickým příkladem může být jádro operačního systému.

- **Kombinace výše popsaných přístupů**

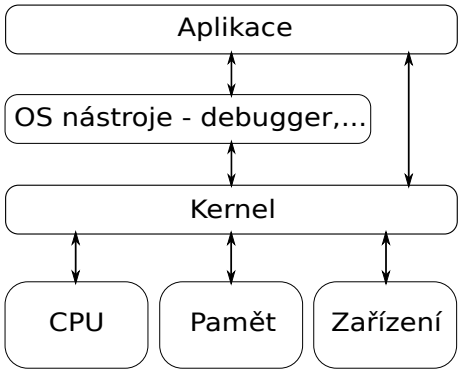
1.2. Operační systém

Většina společností se snaží při vývoji aplikací o co největší výnosnost. A ta je přímo úměrná trvání vývoje. Jak jsme si řekli v předchozí kapitole, jednou z možností je vývoj programů specializovaných na jediný úkol. Takto naprogramovaná malá úloha bude potřebovat jen malé systémové zdroje. Pro jednoduché úlohy je tato cesta vhodná a výhodná. Při řešení komplexnějšího problému budeme potřebovat desítky až stovky jednoduchých procesorů. Rovněž musíme zabezpečit komunikaci mezi nimi. Mnohem výhodnější bude použít jedno výkonnější jádro. Jádro je část procesoru, která provádí výpočty. Pomocí operačního systému (OS) zabezpečíme běh více procesů, tedy vykonávání více úkolů. Použití OS nám poskytne mnoho výhod. Jednoduchý programátorský přístup snižuje náklady na vývoj a současně zvyšuje jeho rychlost. Poskytuje API, zajišťuje přidělování systémových zdrojů procesům s ohledem na priority procesů. Jeho použití ale přináší i úskalí. Režie operačního systému spotřebuje část výpočetního výkonu, který mohl být přidělen procesu. Z toho plyne, že RTS s nejmenším zpožděním a nejlepší odolnosti vůči poruchám bude specializovaný mikrokontrolér se speciálně napsaným, odladěným, programem. Vzhledem k tomu, že probíhá právě jeden proces, nedochází k soupeření o zdroje. Taktéž zdrojový kód bude jasně analyzovatelný. Ve většině případu je takto „promrhaný“ výkon procesorů akceptovatelný, ale při psaní programů pro složité úlohy rádi využijeme pohodlí OS.

Podle Stallings [17] považujeme OS za program, který dohlíží a spravuje běh aplikačních programů. Jde o rozhraní mezi hardwarem a aplikacemi, jak je patrné z obrázku 1.1. Toto rozhraní spravuje zdroje výpočetního stroje. Základní myšlenkou je zjednodušit vývoj a zefektivnit běh aplikační vrstvy. Zkrácený popis služeb poskytovaných OS je podle [17]:

- **Zjednodušení vývoje programů**
  OS poskytuje řadu debugerů a služeb pro odchytávání informací.
- **Obsloužení požadavků programu**
  o spravování zdrojů se aplikace nestará, jen je využívá. Například načtení dat do paměti, inicializace I/O rozhraní, přidělovaní výpočetního výkonu, ...
- **Přístup k periferiím**
  každá periferie má vlastní sadu instrukcí pro obsluhu. Při programování se přistupuje k různým rozhraním stejným způsobem.
- **Kontrolovaný přístup k souborům**
  programátor se nestará o různorodost úložišť (pevný disk, USB disk, pásková mechanika). Rozdílné I/O instrukce a struktura uložených dat je schovaná za jádrem OS. Při přístupu dvou a více procesů k jednomu souboru musí existovat mechanizmus řešení konfliktu. Obdobný mechanizmus je nutný pri přístupu více uživatelů.
- **Přístup k systému**
  Systém musí obsahovat ochranu před nepovolaným přístupem. Neautorizovaným datům a uživatelům musí být odepřen přístup.
- **Detekce a obsluha chyb**
  při běhu systému může docházet k řadám rozličných chyb. Můžeme je rozdělit na hardwarové a softwarové. Kupříkladu selhání periferie, chyba pamětí, dělení nulou, přístup do zakázané oblasti paměti. OS musí eliminovat dopad chyby na bežící aplikace a poskytnout možnost obsloužení chyby aplikacím.

O výše jmenované služby se stará jádro OS, v anglickém jazyce se používá označení kernel.



Obrázek 1.1: Schéma přístupu ke zdrojům v OS

1.2.1. Multitasking

V dnešní době většina OS podporuje běh více aplikací v jednu dobu, multitasking. Jak už bylo zmíněno, tato vlastnost je docílená rychlým přepínání mezi procesy. Pro ujasnění, proces je ekvivalentem aplikace. Způsob přidělovaní zdrojů procesu můžeme rozdělit do dvou základních skupin.

- **Preemptivní multitasking**
  přiděluje zdroje pouze na definované časové kvantum. Posléze je procesu odebere a přidělí dalšímu. O samotné přidělování se stará plánovač úloh. Ten rozhoduje o délce časového kvanta, četnosti změn i pořadí procesů. Jádro odebere zdroje i procesu který přestal fungovat správně. Proto tento zhavarovaný proces neovlivní běh ostatních aplikací. Samotná implementace preemptivního multitaskingu je složitější a má o něco vyšší systémové nároky než kooperativní multitasking. V současné době převládá používaní popisovaného principu. Například Linux, Windows Seven, Mac OS X ...

- **Kooperativní multitasking**
  přidělí zdroje jednomu procesu. Vracení zdrojů je na samotném procesu, a jádro je posléze přidělí dalšímu procesu. Pád jednoho procesu způsobí ve většině případů kolaps celého OS. Tento princip je jednoduchý na implementaci. V jádrech moderních OS se nepoužívá. Je použit například v DOS, RISC OS ...

Všechny data potřebné pro běh úkolu nazývejme kontext. Jedná se o stavy registrů procesoru, obsah pamětí cache, virtuální mapa paměti. Samotné přepnutí kontextu, nutné pro multitasking, je výpočetně náročná operace. Pro rychlejší přepínaní se samotný proces rozdělí na vlákna. Jedná se o jistou odlehčenou formu procesu, která zmenšuje režii při změně kontextu. Tedy i režii multitaskingu. Jeden proces může obsahovat více vláken, minimálně jedno. Samotná vlákna sdílejí jeden paměťový prostor. Komunikace mezi vlákny v rámci procesu je usnadněna, ale pro komunikaci mezi procesy se musí použít speciální mechanizmus.

Pro RT chování OS jsou důležité tři funkce jádra. Plánovač procesů, dispečer procesů a mezi-procesní komunikace. Plánovač procesů rozhoduje kdy mají běžet jednotlivé procesy na CPU. Musí zohledňovat jak priority procesů, tak jejich čekání ve frontě, aby nebyl překročen deadline. Dispečer procesů vykonává potřebné procedury při přepínaní procesů. Jako je uložení potřebných dat pro běh přerušovaného úkolu. Dispečer taky nahrává kontext úlohy která má běžet na procesoru. Mezi-procesní komunikace zabezpečuje přenos dat mezi různými procesy.

Další důležitou vlastností pro RTOS (Realtime operačný systém) je obsluha přerušení. Přerušení můžeme dělit na synchronní a asynchronní. Synchronní produkuje kontrolní jednotka v CPU. Asynchronní jsou produkována připojenými zařízeními v libovolný čas, kdy je potřeba reagovat na výjimečnou událost. Zařízení se snaží zabrat výpočetní výkon pro sebe. Po detekci přerušení systém musí přerušit vykonávanou úlohu. Následně spustí kód určen na obsluhu vyvolaného přerušení. Například se může jednat o stisk tlačítka, dokončení čtení dat z disku, nebo dokončení převodu na A/D převodníku ...

1.2.2. Jádra RT operačních systémů

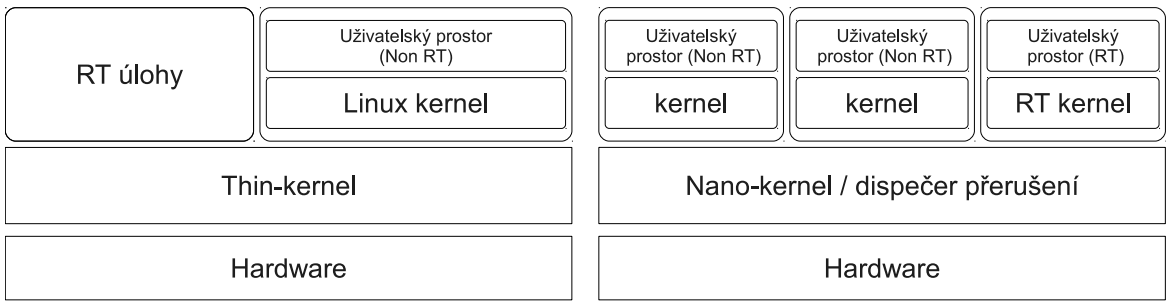
Práce Tim Jones [20] rozděluje Linux jádra pro běh RT úloh na čtyři druhy:

- **Micro-kernel**
  Jedná se o další mikro jádro, viz. obrázek 1.2a, čili abstraktní vrstvu mezi hardwarem a linuxovým jádrem. Které běží jako proces s nižší prioritou. Samotné RT úlohy běží přímo pod tímto novým jádrem, které spravuje příchozí přerušení. Linuxové jádro nedokáže přerušit běh programů v mikro jádře. Negativem je ztráta plné platformové podpory pro ne RT procesy běžící pod standardním jádrem. Vyvážením je, že při použití tohoto přístupu, úlohy běžící jako RT splňují hard RT požadavky. Příkladem můžou být projekty RTAI, Xenomai, RTLinux.

- **Nano-kernel**
  Schematické zobrazení přístupu je na obrázku 1.2b. Jedná se o minimalizovaný micro-kernel. Přerozděluje výpočetní zdroje pro více OS a rozhoduje o prioritách pro tyto vyšší vrstvy. V dnešní době se tento termín nepoužívá. Nahradil ho termín micro-kernel. Jelikož moderní mikro jádra jsou dostatečně výkonná a nahrazují nano jádra. Příkladem je projekt ADEOS.

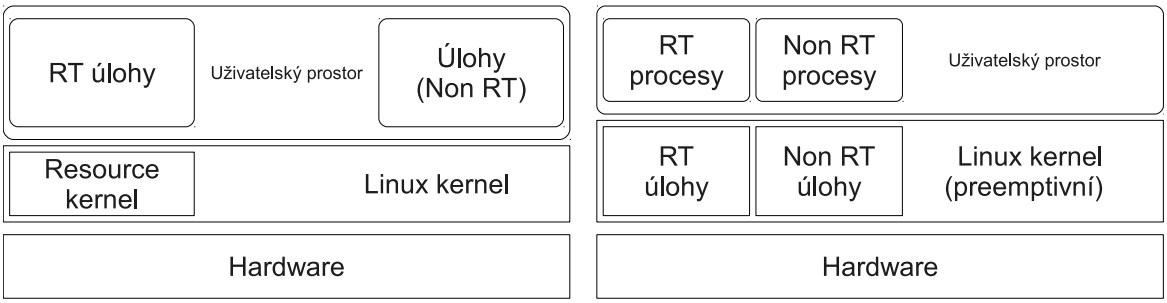
- **Resource-kernel**
  V tomto přístupu se do jádra přidá modul, který zarezervuje definovaný výpočetní výkon. Znázornění vidíme na obrázku 1.3a. Velikost rezervovaných zdrojů se odvíjí od parametrů pro danou RT úlohu. Jako je deadline, perioda zpracování či předpokládaná spotřeba cyklů při obsluze. Samotný modul má vlastní API, pro vyjednávání těchto podmínek. Po definici požadavků se snaží alokovat systémové zdroje. V úspěšném případě garantuje procesu vyjednané podmínky. V neúspěšném ohlásí chybu a RT proces nespustí. Projekt Linux/RK je příkladem použití toho jádra.

- **Standard kernel**
  Přístup při použití standardního kernelu vidíme na obrázku 1.3b. Dodržování RT požadavků řeší přímo jádro, které přiděluje zdroje. Změny v jádře potřebné k dodržování RT požadavků popíšeme v 1.3



(a) Mikro-kernel (b) Nano-kernel

Obrázek 1.2: Přístup k RTOS při použití micro-kernel a nano-kernel



(a) Resource-kernel (b) Linux kernel

Obrázek 1.3: Přístup k RTOS při použití resource-kernel a linux kernel

1.2.3. Nevýhody realtime operačních systémů

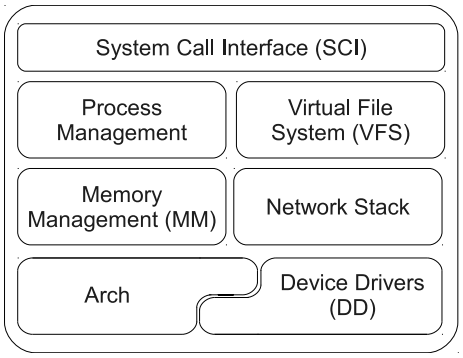
Použití RTOS nemusí vždy přinášet užitek. Ovšem při řízení, zpracovávání, dějů s krátkou časovou konstantou je nutností. Není to z důvodu rychlejších reakcí, ale standardní OS mají v nejhorších případech příliš vysoké latance. Jak bylo vzpomenuto v kapitole 1.2. Samotná RT úprava jádra musí mít z podstaty věci horší propustnost dat a nižší výpočetní výkon. Časté přepínaní mezi běžícími procesy přináší nárůst režie jádra. S narůstajícím počtem RT procesů tato režie prudce stoupá. Použití na serverech, kde se očekává veliká propustnost dat může mít fatální následky. Ale pro většinu stolních počítačů přináší rychlejší odezvy. Je potřeba zvážit použití RTOS a také jeho nastavení. Jelikož platí přímá úměra mezi předvídatelností a zkonzumovaným výpočetním výkonem režií jádra.

1.2.4. Operační systém Linux

Patří do rodiny UNIXů, kde například patří: HP-UX, FreeBSD, Solaris, QNX, Mac OS X, ...Vývoj Linuxu začíná Linus Torvalds. Při studiích na Helsinské univerzitě se potkává s UNIX-like operačními systémy. Jedním z nich je výukový MINIX, kterým se nechal inspirovat. Programovat vlastní operační systém začíná na nově zakoupeném počítači s procesorem 386. První verzi linuxového jádra s označením 0.01 vydal v září roku 1991 [8]. Zdrojový kód je napsán v jazyce C. Základní pohled na jednotlivé části jádra je na obrázku 1.4. Jen minimální část, platformově závislá, je napsána v assembleru. Zásadní změnou vůči ostatním UNIX-like systémům je jeho otevřenost. Zdrojové kódy se šíří pod licencí GPL. Zkráceně, licence zabraňuje komukoliv omezovat distribuční práva a přikazuje zpřístupnit zdrojové kódy. Brzy po vydání vzniká kolem jádra celosvětová komunita, která dobrovolně pracuje na vývoji jádra. Hlavní strom vývoje nese pojmenování vanilla. Dokumentační projekt [10] charakterizuje další důležitou vlastnost linuxového jádra, a ta je dodržování standardů POSIX-1 a POSIX-2, vydávané institutem pro standardizaci IEEE. Tento rys zjednodušuje přenositelnost již napsaných aplikací na různé platformy i mezi odlišnými UNIX-like systémy. Samotná norma POSIX vzniká kolem roku 1988 podle rozhraní UNIX. Popisuje základní systémovou funkcionalitu a řadu různých volitelných součástí. Jádro rozděluje systém do dvou režimu:

- **uživatelský režim (user space)**
  je vlastně prostor pro běžné uživatelské aplikace. Jde o neprivilegovaný režim, kde aplikace mají přístup jen do svého paměťového prostoru. Kdežto jádro přístup do této oblasti má. Samotné aplikace nemohou využívat některá systémová volání.

- **režim jádra (kernel space)**
  jde o privilegovaný režim. Jedná se o paměťový prostor samotného jádra, kde aplikace nemají přístup.



Obrázek 1.4: Jednoduchý strukturální pohled na linuxvé jádro,převzato z [19]

V dnešní době je jádro portováno na většinu architektur, například PowerPC, ARM, AVR32, Alpha, SPARC a další. Ať už jsou určené pro servery, stolní počítače či embedded systémy. Název Linux se vžil pro označení všech distribucí na bázi tohoto jádra a projektu GNU. Až tento celek můžeme označovat jako operační systém. Cílem tohoto projektu je vytvořit kompletní soubor pomocných programů-nástrojů pro fungování OS podobného Unixu. Jedná se například o gcc, bash, autoconf ... Samotný balík programů GNU není omezen jádrem Linuxu. Může pracovat i s jinými jádry. V práci bude použito spojení GNU/Linux i samotného kompilátoru gcc (GNU Compiler Collection). Samotné monolitické jádro je preemptivní s podporou načítaní externích modulu.

Licenční podmínky mají pro firemní použití výhodnou i nevýhodnou stránku. Pořizovací cena je nulová. Otevřenost zdrojového kódu usnadňuje úpravy. Jenže je nutné zpřístupnit zdrojové kódy úprav. Což může být v rozporu s firemní politikou.

1.3. Linux a realtime

V nepreemptivním jádře není možné přerušit systémové volání z uživatelského prostoru. A to i případě potřeby přerušení procesu s nízkou prioritou. Preemptivní jádro je v případě procesu s vysokou prioritou schopno přerušit systémové volání procesu s nižší prioritou. V standardní verzi Linuxového kernelu, při zapnutí volby `PREEMPT=y` v konfiguraci se jádro stane preemptivním. Na architektuře x86 (stolní počítače) je dále možné zapnout podporu časovače s vysokým rozlišením `HPET=y`. Zde již můžeme pracovat s rozlišením pod 1μs [20]. To stačí pro splnění požadavků pro značnou část soft RT úloh. Na architektuře ARM získáme časovač s vysokým rozlišením po zapnutí volby `HIGH_RES_TIMERS=y`. Samotné konfigurování kernelu je vysvětleno v kapitole 3.

Další úprava spočívá v aplikování RT rozšíření, RT patch. V klasickém linuxovém jádru bylo potřeba vykonat několik úprav, aby bylo jádro možné použít pro hard RT aplikace.

1.3.1. Plánovač v kernelu

Pořadí běžících vláken určuje část jádra zvaná scheduler-plánovač. Určení priority není zcela triviální záležitostí, jak by se mohlo zdát. Každé přepnutí vlákna je výpočetně náročné a proto není žádoucí takto mrhat časem. Plánovač se snaží o minimalizaci změn kontextu, která jsou nepřímo úměrná efektivitě. Protichůdný požadavek klade co nejnižší zpoždění odezvy. Tady je snaha o co nejčastější přepínání mezi vlákny. Aktivní vlákna můžeme rozdělovat podle povahy jakou zatěžují systém [1]. A to vlákna zatěžující CPU a vlákna které při běhu využívají I/O zařízení. Druhé jmenované značnou část času tráví čekáním na dokončení požadované I/O operace. Druhou variantou je rozdělení do tří skupin podle požadavků na rychlost odezvy [1].

- **Interaktivní procesy**
  jsou neustále ovlivňovány uživatelem. Například se jedná o stisk klávesnice nebo pohnutí myší. Průměrná odezva musí být v intervalu 50 ms až 150 ms.

- **Dávkové procesy**
  nemají žádnou interakci s uživatelem. Úloha vetšinou běží na pozadí. Kupříkladu kompilace programu.

- **Realtime procesy**
  kladou vysoké nároky na plánování. Nesmějí být blokovány procesy s nižší prioritou a požadují dostatečně krátké a stabilní zpoždění.

Každý proces je plánován podle jedné z plánovacích tříd. Každá třída se řídí vlastní pravidly. Nejčastější používané třídy jsou [7]:

- **SCHED_NORMAL**
  Pro proces v této plánovací politice je nemožné dostat realtime prioritu. Má jednu úroveň statické priority a 40 úrovní dynamické priority. které se dají nastavit příkazem `nice`. Plánovač mění přidělované časové kvantu na základě historie využívání procesoru úlohou. Plánovač je spravedlivý a k zpracování se dostane i úloha s nejnižší prioritou.

- **SCHED_BATCH**
  plánovač pro dávkové úlohy, nepředpokládá interakci úlohy a uživatele. Jde o modifikaci předchozího plánovače. Při výpočtu dynamické priority se úloha vždy posuzuje, jakoby neustále požadovala běh na procesoru. Tedy je permanentně penalizována.

- **SCHED_FIFO**
  prvně příchozí proces je jako první odbaven. Plánovač používá 99 úrovní priorit. Jde o realtime plánovač. Úlohy s nižší prioritou a také úlohy s normální plánovací politikou jsou ignorovány a zdroje jsou přiděleny úloze s nejvyšší prioritou.

- **SCHED_RR**
  jádro přidělí procesu časové kvantum, během kterého je proces zpracováván. Velikost časových kvant se počítá z priority procesu, kterých je také 99. Rovněž se jedná o realtime plánovač. Na rozdíl od předchozí plánovací politiky, tady časové kvanta rotují mezi procesy se stejnou prioritou. Nečeká se na ukončení první obsloužené úlohy.

Existuje spousta dalších tříd a jejich modifikací. Pro RTOS je zajímavý plánovač EDF (Earliest Deadline First), který plánuje na základě dedaline úloh. U víceprocesorových systémů musí plánovací algoritmus rozdělovat úlohy na jednotlivé jednotky CPU. Používají se tři základní modely. Symetrické rozdělení na každý procesor. Asymetrické rozdělení a Non-Uniform Memory Access. Kde každý procesor má vyhrazenou vlastní paměť.

Kromě politiky plánování je každému procesu přidělena také jeho priorita. Kterou může, ale nemusí daný plánovač zohledňovat. Zobrazení priorit a plánovací politiky všech bežících procesů vypíše příkaz:

```
ps -e -o pid,comm,pri,rtprio,policy
```

U procesů se standardním plánovačem SCHED_NORMAL je priorita od 0 do 39. Úloha s normální prioritou má hodnotu 20. Směrem k číslu 39 priorita procesu klesá a směrem k nule se zvyšuje. RT priorita je číslované od 1 do 100, kde vyšší číslo značí vyšší prioritu.

1.3.2. Úpravy v RT patch

Úpravy na kernelu z hlavního stromu, pro potřebu RTOS, můžeme charakterizovat třemi body.

- **Plánování dle priorit**
  RT úloha s nejvyšší prioritou musí přerušit běh ostatních úloh. Propustnost takto upraveného systému se snižuje, ale zároveň se také snižuje zpoždění.

- **Zmenšení chvění**
  v normálním systému se příchozí přerušení zpracovávají přednostně, i když nemusí být důležité. Z tohoto důvodu mají reálně zpoždění pro jednu úlohu veliký rozptyl. Cílem je tento rozptyl minimalizovat. Například zmenšením počtu nepřerušitelných částí.

- **Zvýšení determinovanosti**
  je žádoucí přesně předvídat maximální zpoždění.

Několik úprav bylo provedeno v meziprocesorové komunikaci. Jejich cílem je implementovat přerušitelné synchronizační primitiva, dědění a inverzi priorit, soft přerušení.

Mějme základní synchronizační primitiva jako je mutex, spinlock, semafor, barrier. Popis a použití jednotlivých typů primitiv, může zvídavý čtenář najít v publikaci [4]. Oblast kódu mezi získáním a uvolněním zámku nad zdrojem nazývejme kritická sekce. Kód v takovéto oblasti se bude provádět pouze jediným vláknem. Snaha o minimalizaci kritických sekcí by měla být v samotném návrhu RT aplikace. První změnou bylo nahrazení nepřerušitelného spinlock za přerušitelný zámek rt_mutex. Základní synchronizační primitiva byla rozšířena o rt_mutex, kde je implementována metoda priority inheritance [14]. Mějme dva procesy C,D které soupeří o zdroj. Proces C, který má nižší prioritu než proces D, uzamkne zdroj. Jeho vykonávaní je přerušeno procesem D, ale ten požaduje uzamčení zdroje. Aby se nemuselo čekat na dokončení všech procesů s vyšší prioritou než má proces C, tak mu bude přidělena nejvyšší priorita z procesů soupeřících o daný zdroj. Po jeho dokončení a odemčení zdroje, bude pokračovat ve vykonávání kódu proces D. Tato metoda je implementována v synchronizačních semaforech.

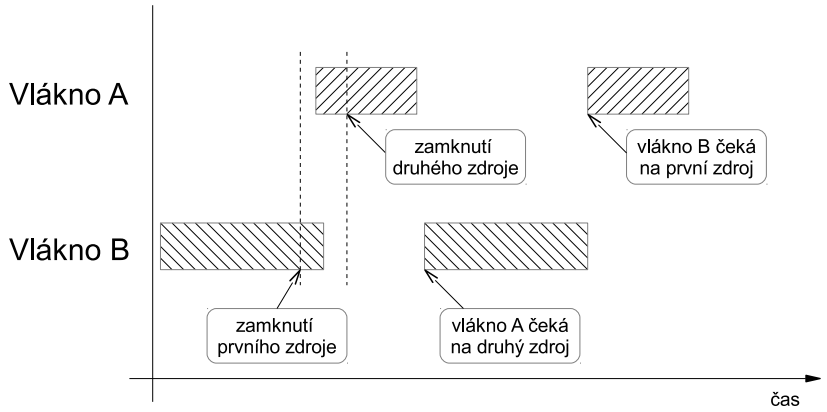
Další změnou byly úpravy ve zpracování přerušení. Typické zpracování hardwarového přerušení je následující:

1. Periferie vyvolá požadavek na přerušení.
2. Tento signál se dostává na řadič přerušení, který jej přijme a zpracuje.
3. Řadič přerušení generací signálu informuje procesor, pomocí speciálního registru, o požadavku přerušení od periferie.
4. Vypnou se všechny přerušení, aby nebylo možné pozastavit proceduru pro obsluhu přerušení.
5. Uloží se kontext bežící úlohy a spustí kód pro obsluhu přerušení.
6. Procesor přečte z řadiče číslo typu IRQ. Když není maskované, nebo vypnuté, odbaví ho asociovaným ISR.
7. Vynulování příznaku pro obsluhovanou periferii.
8. Zapnutí přerušení.
9. Procesor pokračuje ve zpracovávání přerušené úlohy.

Popsáno bylo hardwarové přerušení, také existují softwarové přerušení, kde proces žádá o obsloužení. Samotné IRQ může reprezentovat RT událost, která se musí odbavit do stanoveného deadline. Proto je pro RTOS důležitá rychlá reakce na příchozí IRQ, tedy zjišťování stavu příchozích IRQ musí probíhat co nejčastěji. V popsaném modelu je po celou dobu obsluhy IRQ nemožné vyvolat nové přerušení. RT úprava přerušení spočívá ve změně ISR (samotná obsluha požadavku) na přerušitelná vlákna, nazývejme je soft IRQ. Takto se převedla většina obsluh přerušení. Po této změně je možné obsluhu IRQ s nízkou prioritou přerušit při příchodě IRQ s vyšší prioritou. Také v případě čekání více přerušení na odbavení se upřednostňuje to s vyšší prioritou. Proces obsloužení přerušení se rozděluje. Při vzniklém přerušení s nízkou prioritou se vykonají jen nezbytně nutné úkony a dokončení ISR se naplánuje na později.
Způsoby hlášení hardwarového přerušení můžou být následující [7]:

- **hlášení úrovní (level-trigered)**
  zařízení nastaví určitou úroveň signálu a ponechá ji nastavenou až do obsloužení.
- **hlášení hranou (edge-trigered)**
  zařízení zašle impulz. Důležitá je hrana (náběžná, sestupná), nikoli úroveň.
- **hybridní**
  jedná se o hlášení hranou, kde se po stanovenou dobu drží nastavená úroveň signálu. Používá se pro některá důležitá přerušení.
- **hlášení zprávou (message-signaled)**
  žádost o přerušení se neposílá signálovým vodičem, nýbrž zprávou po sběrnici (PCI Express).

Při soupeření o zdroje dvou a více vláken může nastat uváznutí v nekonečném čekání, čili deadlock. Příklad uváznutí obou dvou vláken znázorňuje obrázek 1.5.



Obrázek 1.5: Znázornění uváznutí typu deadlock

Obdobné riziko nastává při implementaci uvolnění zámku při neúspěšném zamknutí druhého zdroje. V tomto případě dojde k uváznutí typu livelock. Kde obě vlákna skončí v nekonečné smyčce uvolňování a zamykání zdrojů. Tato uváznutí není možné systémově odstranit. Proto představují značné riziko v RTOS.
Tabulka 1.1 srovnává latence na různě upravených jádrech verze 2.6.26.

Linux	**Standard**	**Preempt**	**RT Patch**
průměr	10,94μs	11,58μs	12.98μs
maximum	4656,44 μs	5623,92μs	104.06μs

Tabulka 1.1: Porovnání zpoždění Linuxových jader pomocí LPP test, zdroj [12]

1.3.3. Zpoždění

V informatice můžeme zpoždění definovat jako časový interval mezi vznikem požadavku a začátkem spuštění obslužné rutiny. Délka obslužné rutiny je variabilní a je věcí samotného návrhu aplikace. Většina požadavků je reprezentována přerušením, softwarovým nebo hardwarovým. Celkové zpoždění vzniklé při přerušení můžeme rozdělit následovně:

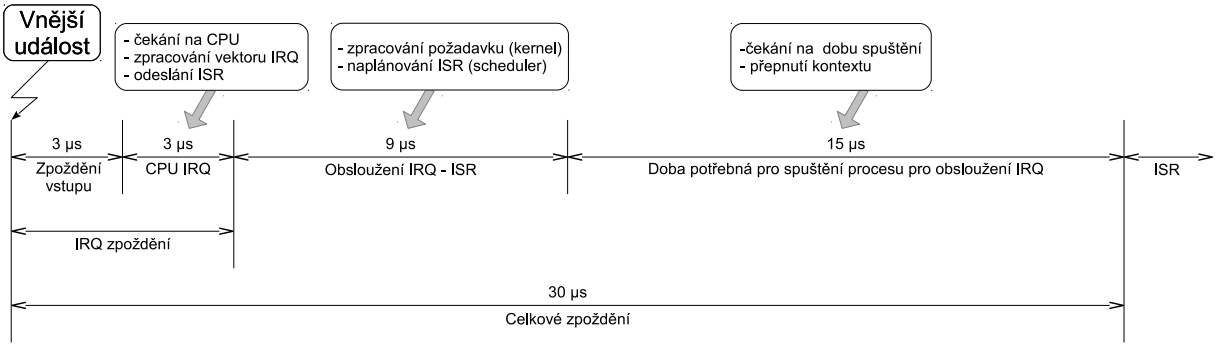
1. Po obdržení signálu přerušení vzniká zpoždění na samotném vstupu-řadiči přerušení.
2. Další čas spotřebuje CPU než zjistí existenci IRQ a zpracuje ho. Z vektoru přerušení asociuje ISR pro vzniklé IRQ.
3. Po vybrání asociované ISR se musí tato úloha spustit. V momentu spuštění již mluvíme o odpovědi na vstupní signál. Tedy nečekáme na ukončení ISR, jelikož ta může být specifická. Toto zpoždění můžeme nazvat latence plánování. Používaný termín je scheduling latency. Jedná se o stav, kdy proces s nejvyšší prioritou je připraven běžet a nějakou nenulovou dobu trvá než dostane přidělené prostředky. Velikost této doby značně ovlivňuje tzv. peemption latency.

Na obrázku 1.6 vidíme toto rozdělení i s popisem časové osy. Tato celková doba zpoždění odezvy se nazývá interrupt latency. Obě tyto zpoždění nebudou mít stálou hodnotu.

Budou kolísat kolem průměrné hodnoty. Tato nechtěná proměnlivost se nazývá chvění - jitter.

Při periodických úlohách můžeme mluvit o wakeup latency, tedy zpoždění vzniklé při probouzení úlohy. Jde o rozdíl mezi požadovaným časem spuštění a reálným časem spuštění. Jediným ovlivňujícím faktorem na velikost wakeup latency je fungování plánovače procesů. Proto se ve většině literatur termín scheduling latency ztotožňuje s termínem wakeup latency [16].

Na různé možnosti měření zpoždění a porovnávaní výkonnosti RTS se blíže podíváme v kapitole 4.

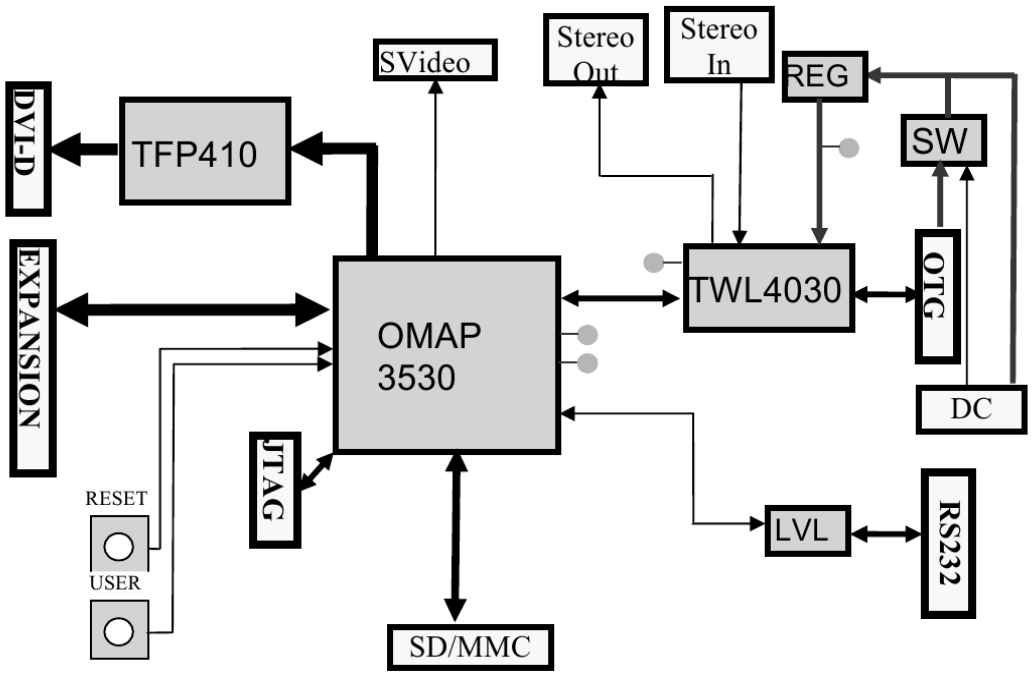


Obrázek 1.6: Zpoždění při IRQ

2. BeagleBoard

Pro testovací účely byl vybrán kit BeagleBoard verze B7. Jedná se o opensource projekt s cílem dodávat kity v nízké cenové relaci. Pro snížení výrobních nákladů se autoři rozhodli desku osadit jen tím nejnutnějším vybavením. I některá rozhraní pro připojení periferií nejsou osázené. Podle manuálu [9] není v desce implementována plná podpora pro vývoj. Ale je snaha o využití všech standardních rozhraní pro připojení periferií. Blokové schéma zapojení je viditelné na obrázku 2.1. Použitý je mikroprocesor OPAM3530 od Texas Instrument, z rodiny ARM Cortex-A8. Kit je napájen pomocí 5V externího zdroje přes 5.5mm konektor. Minimální proudová zatížitelnost napájecího zdroje má být 500mA a maximální proudový odběr kitu je 2A. Druhá možnost napájení je přes USB OTG konektor přímo z USB rozhraní počítače, jehož funkčnost jsme ověřili. Při této možnosti byla nutnost mít zapnutý počítač. Z tohoto důvodu byl jako zdroj použit adaptér na 5V s maximální zatížitelností 3A.

Instalované jsou dva typy paměti. První typu NAND a to o velikosti 256MB. Druhá je typu SDRAM o kapacitě 128MB a taktu 166MHz. Obě paměti jsou připojeny metodou POP, tedy na čelo procesoru. Z tohoto důvodu nenajdeme na žádném integrovaném obvodu na desce nápis OMAP3530. Další paměťová média můžeme připojit pomocí SD/MMC a USB rozhraní. V práci byla používaná 2G SD karta.



Obrázek 2.1: Blokové schéma zapojení kitu, převzato z [9]

2.1. OMAP3530

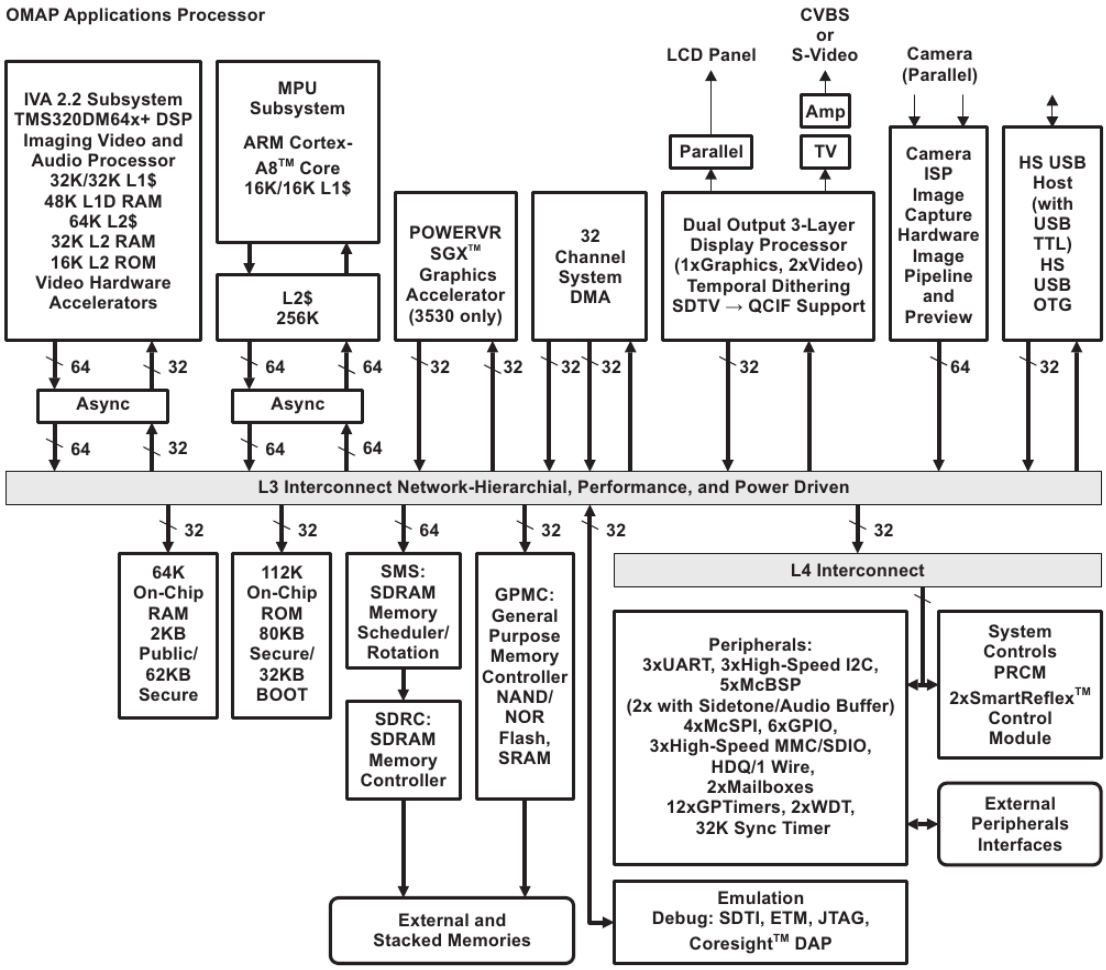
Jak již bylo zmíněno, procesor patří do rodiny ARMů. Kterou začala vyvíjet britská firma ARM Limited v roce 1984 pro společnost BBC. Samotné mikroprocesory ARM mají architekturu RISC (Reduced Instruction Set Computers) s redukovanou instrukční sadou. Ta má za následek snížený počet tranzistorů při implementaci jádra, který následně vyúsťuje v nízkou energetickou náročnost. Což je jedna z předností čipů ARM. Malá instrukční sada procesoru také značně zjednodušuje vývoj překladačů a zvyšuje efektivnost překladu i optimalizací. Samotná firma nabízí i procesory s označením Cortex-R, jedná se o embedded procesory specializované pro RTS [2]. V práci jsme se věnovali procesoru Cortex-A8,který je optimalizován pro aplikační použití.
Pozastavme se u číslování verzí jednotlivých architektur procesorů ARM, které je lehce matoucí. Proto je jejich přehled uveden v tabulce 2.1. Architekturu jádra je důležité znát při výběru OS a využití tzv. cross-compilace. Tedy kompilace spustitelných souborů pro jednu architekturu, v našem případě ARM. Na architektuře jiné, například x86.

Architektura	**Rodina procesorů**
ARMv1	ARM1
ARMv2	ARM2, ARM3
ARMv3	ARM6, ARM7
ARMv4	StrongARM, ARM7TDMI, ARM9TDMI
ARMv5	ARM7EJ, ARM9E, ARM10E, XScale
ARMv6	ARM11, ARM Cortex-M
ARMv7	ARM Cortex-A, ARM Cortex-M, ARM Cortex-R

Tabulka 2.1: Přehled verzí jader ARM.

Licenční politika umožňuje zakoupit licenci pro použití jádra ARM ve vlastních integrovaných obvodech. Toho využívá i firma Texas Instrument (dále TI) v mikroprocesoru OMAP3530. Kde na jednom čipu je implementováno jádro ARM Cortex-A8 a přímo k němu připojené periferie, grafický akcelerátor, DSP (Digital Signal Procesor) akcelerátor a akcelerátor pro zpracování obrázku, videa a zvuku - IVA2.2. Tyto součásti připojené k jádru jsou již vyvinuty společností TI. Celé blokové schéma použitého procesoru je vidět na obrázku 2.2. Kvůli implementaci všech potřebných součástí pro daný účel na jeden integrovaný obvod, se pro tento druh mikroprocesorů používá zkratka SoC (System on Chip).

Procesor pracuje na frekvenci 600MHz a obsahuje 16 kB I-cache, 16 kB D-cache, 256 kB L2 cache. Podle [9] dosahuje 1200 Dhrystone MIPS (milionů instrukcí za sekundu). Výsledkem testu Dhrystone je číslo vyjádřující počet iterací měřící smyčky. Samotný test se snaží reprezentovat zátěž široké škály softwarů, napsaných v různých programovacích jazycích, pracujících v celočíselném režimu.
Jádro je superskalární s podporou OpenGL ES 2.0. Také obsahuje koprocesor NEON pro urychlení zpracování multimediálních úkolů. Grafický akcelerátor by měl být schopen zobrazovat deset milionů polygonů za sekundu.



Obrázek 2.2: Blokové schéma procesoru OMAP3530, převzato z [11]

BeagleBoard podporuje následující funkce procesoru.

- POP Paměťové rozhraní
- 1Gb MDDR (128Mbytes)
- 2 Gb NAND flash (256 MB)
- 24 bit RGB rozhraní displeje (DSS)
- SD/MMC
- USB OTG
- NTSC/PAL/S-Video výstup
- správu napájení
- sériové rozhraní
- I2C rozhraní
- I2S audio rozhraní (McBSP2)

- rozšíření McBSP1
- JTAG rozhraní pro ladění

2.2. Rozhraní na BeagleBoard

Na desce se nachází následující rozhraní:

- JTAG konektor
- DVI-D výstup
- S-video výstup
- vstup a výstup pro audio signál. Jedná se o dva standardizované 3.5mm konektory.
- USB OTG konektor
- SD/MMC+ konektor. Podporuje také připojení externích zařízení jako WiFi, video kamer, GPS modulů. Nabootování z paměťové karty je možné jen u 3V karet.
- RS232 IDC-10 konektor
- 28 kolíkový konektor pro připojení rozšíření

Blíže se budeme věnovat posledním dvou rozhraním. Pro připojení k modulu budeme používat sériový port. Ten je vyveden pomocí IDC-10 konektoru. K správnému připojení na konektor DE-9, standardně osazen na většině personálních počítačů, nám poslouží zapojení uvedené v příloze A.1. K správné komunikaci je potřeba nastavit parametry sériového přenosu , které jsou:

```
BAUD RATE - 115200
DATA - 8 bit
PARITY- none
STOP - 1bit
FLOW CONTROL - none
```

Ke komunikaci s kitem v práci použijeme program minicom. Úplně stejnou službu nám poskytne jakýkoliv z jiných terminálových programů (screen, Tera Term, Hyperterminal, PuTTy, ...).

Rozhraní pro rozšíření je možné variabilně nakonfigurovat pomocí přepínačů v mikroprocesoru. Jednotlivé možnosti jsou znázorněné v tabulce 7-4 na straně 774 v [11]. V poli „Register Name“ nacházíme položku odpovídající pinu podle schématu v příloze A.2. Jednotlivé módy zobrazují možné konfigurace pinu. Nehledě na režim je možné každý pin nastavit do jednoho ze dvou režimu: vstupní a výstupní. Způsob nastavování režimu popíšeme v následující kapitole 3.1.4.

Na desce se nachází dvě tlačítka: user, reset. Jak je uvedeno v [9]: Jednou z hezkých vlastností procesoru OMAP3530 je, že může načítat systém přímo z SD/MMC paměťové karty. Stačí když podržíme tlačítko USER a přepneme tlačítko RESET.

3. Práce s BeagleBoard pod Linuxem

Pro rychlý test funkčnosti kitu BeagleBoard stačí jeho připojení k napájecímu napětí. Okamžitě se rozsvítí LED dioda PWR. Po připojení terminálovým programem na sériové rozhraní, můžeme zkusit načíst systém již popsanou procedurou v části 2.2. Je-li systém nahraný v paměti NAND, začne se načítat. V opačném případě vidíme po každém promáčknutí přibývající řetězec `40V`. V následující části si v krátkosti vysvětlíme vlastní instalaci Linux/GNU na BeagleBoard. Samotná kapitola nemá za účel vysvětlovat všechny možnosti použitých příkazů a dopodrobna rozebírat použité přepínače. Na obdobné otázky odpoví manuál příkazů. Stačí spustit `man 'pozadovany prikaz'`.

3.1. Instalace Linuxového systému

V textu budeme pracovat na architektuře x68_64 na linuxové distribuci Gentoo verze 10.0. Když nebude uvedeno jinak, pracujeme s právy superuživatele (root). Také předpokládáme podporu paměťových karet a používaných souborových systémů v používaném jádře. Běžná distribuční jádra těmto požadavkům vyhovují. Samotná instalace začíná vytvořením překladače a příslušných utilit potřebných pro překlad na architekturu armv7. Použijeme kompletní soubor nástrojů pro programování vytvořených pod GNU. Vžité pojmenování toolchain. S nově vytvořeným překladačem můžeme dále pracovat na sestrojení samotného jádra. A také ho využít při cross kompilaci na osobním počítači.

Pro rychlou orientaci uvedeme některá specifika vybrané distribuce Gentoo. Jedná se o komunitní distribuci založenou na kompilovaní zdrojových kódů. Samotný balíčkovací systém se nazývá Portage, jenž řeší závislosti balíčků a nastavuje jim konfigurační parametry při kompilaci. Informace o jednotlivých balíčcích jsou uloženy v adresáři, jeho cesta je definována v souboru `/etc/make.conf`. Kde jsou také uloženy standardní parametry pro kompilovaní. Samotné přidávání balíčku můžeme provést příkazem ebuild, který je nízkoúrovňový. Nebo námi preferovaným příkazem emerge. Ten automaticky řeší potřebné záležitosti potřebné k instalaci. Například se jedná o vyřešení závislostí k jednotlivým balíčkům, stažení zdrojových kódu, kompilaci, instalaci do systému ...

3.1.1. Toolchain pro architekturu armv7

Pro zhotovení toolchain, pro zvolenou architekturu, s výhodou využijeme předpřipravené skripty v distribuci Gentoo. Jedná se o balíček sys-devel/crossdev.

```
emerge sys-devel/crossdev
```

Samotný balíček skriptů crossdev přidává záznam do stromu balíčků Portage. Není žádoucí, aby se míchal do standardního stromu na hostitelském systému. Proto si vytvoříme místní repositář balíčků. V souboru `repo_name` je uložen název repositáře, který budeme vidět ve výstup programu emerge.

```
mkdir -p /usr/local/portage/profiles
echo "local_overlay" > /usr/local/portage/repo_name
```

Po následující úpravě souboru `/etc/make.conf` zná systém novou cestu k repositáři. Když nepoužíváme nástroj layman, pro spravování externích repositářů, můžeme první řádek vynechat. Skript crossdev ukládá balíčky do druhého repositáře uvedeného v proměnné `PORTDIR_OVERLAY`.

```
source /var/lib/layman/make.conf
PORTDIR_OVERLAY="/usr/portage␣/usr/local/portage␣$PORTDIR_OVERLAY"
```

Nyní můžeme spustit instalaci toolchain pro cílovou architekturu armv7.

```
crossdev -S -oS /usr/portage/ /usr/local/portage/ local \
-t armv7a-unknown-linux-gnueabi --ex-gdb
```

vysvětlení použitých parametrů:

-S	skript má použít stabilní verze balíčků
-oS	určuje repositáře, které se budou prohledávat
-t	definuje cílový stroj armv7a-unknown-linux-gnueabi řetězec je ve formátu architektura-platforma-OS-knihovna C
- -ex-gdb	oznamuje skriptu, že má instalovat ladící nástroj GNU

Skript automaticky nainstaluje následující balíčky:

- binutils – Nástroje pro práci s binárními soubory.
- gcc – kolekce nástrojů GNU pro kompilaci.
- glibc – GNU knihovna C.
  Změnou volby je možné instalovat i uclib, jedná se o specializovanou knihovnu pro embedded systémy.
- linux-headers – hlavičkové soubory jádra potřebné pro kompilaci knihovny C.
- gdb – ladící nástroj GNU.

Jelikož v této práci nebudeme využívat kompilaci balíčků pomocí emerge pro architekturu armv7 na hostovacím systému. Proto další konfiguraci popsanou na stránkách Gentoo [3] vynecháme.

Posledním krokem je vytvoření pomocného skriptu pro zjednodušení kompilace s použitím nástroje make. Skript předává kompilátoru parametry o cílové architektuře, kterou jsme vytvořili skriptem crossdev. Přepínačem `-j5` určujeme počet spuštěných simultánních kompilací. Jeho hodnota je určena hardwarovým vybavením hostitelského počítače. Dále všechny vstupní parametry budou předány nástroji make.
Tedy vytvoříme soubor `/usr/bin/armv7a-unknown-linux-gnueabi-make` s následujícím obsahem:

```
#!/bin/bash
make -j5 ARCH="arm" CROSS_COMPILE="armv7a-unknown-linux-gnueabi-" $*
```

Tento soubor změníme na spustitelný pro všechny skupiny, po provedení příkazu:

```
chmod +x /usr/bin/armv7a-unknown-linux-gnueabi-make
```

Kompilaci můžeme také provést bez pomoci crossdev, podle nesčetných návodů na Internetu.

3.1.2. Kompilace kernelu pro architekturu armv7

Aby bylo možné zavést systém z SD paměťové karty je nutné ji upravit. Proto ji upravíme před samotnou kompilací jádra pro BeagleBoard. Úpravu provedeme pomocí programu `fdisk`. Pro ilustraci předpokládejme že SD karta je reprezentována souborem `/dev/sdc`. Spustíme příkaz `fdisk /dev/sdc`. V normálním režimu vytvoříme prázdnou tabulku rozdělení pro systém MS-DOS. V expertním režimu nastavíme 255 hlav, 63 sektorů a počet cylindrů vypočítáme z kapacity paměťové karty pomocí vzorce:

$$N_{cylindru} = \frac{pocet\ bajtu}{N_{hlav} N_{sektoru} 512\ bajtu} \tag{3.1}$$

Celkový počet bajtů zjistíme vypsáním aktuálního rozdělení karty. Vzorec 3.1 určuje počet bajtu na cylindr, požadujeme 512 bajtů. Je-li výsledné číslo desetinné, pak tuto desetinou část zanedbáme. Vrátíme se do normálního režimu, kde vytvoříme první oddíl typu FAT32. Velikost můžeme volit dle vlastních požadavků, například 64 MB. Nově vytvořený oddíl označíme jako startovací. Volba dalšího rozdělení paměťového prostoru karty je z hlediska zavádění systému irelevantní. Tedy ji upravíme dle vlastních požadavků. My jsme zvolili 1.4 GB pro samotný systém a zbylých 369 MB pro repositář balíčků. Následně tabulku rozdělení uložíme a můžeme na jednotlivých oddílech vytvořit souborové systémy.

Pro zavedení systému je nutné první oddíl naformátovat na souborový systém typu FAT32. My jsme ho označili jmenovkou „boot“. Je třeba zdůraznit, že je nutné na tento zaváděcí oddíl jako první zkopírovat binární soubor MLO, viz. část 3.1.3. Pro další oddíly jsme zvolili typ EXT3. Kořenový systém je naformátován standardními parametry a s jmenovkou „gentoo“. Poslední oddíl, pro repositář, jsme naformátovali s pozměněnými parametry. Jelikož v repositáři je uloženo mnoho souborů s relativně malou velikostí. Proto jsme zvolili velikost bloků, i počet bajtů na jeden i-uzel, na jeden kB. Dále jsme jej označili jmenovkou „portage“ a nastavili velikost rezervace pro superuživatele na nula procent. Tedy spustíme následující příkazy:

```
mkfs.vfat -F 32 -n boot /dev/sdc1
mkfs.ext3 -L gentoo /dev/sdc2
mkfs.ext3 -L portage -b 1024 -i 1024 -m 0 /dev/sdc3
```

Příloha B nás provede základy nezbytnými pro instalaci jádra. V práci je použito jádro verze 3.2.14 s RT patchem verze 24. Tedy celkové označení je 3.2.14-rt24. Jedná se o jádro z hlavního stromu. Nebylo vybráno jádro se speciálními úpravami pro ARM, jelikož jeho vývoj setrvává v minor verzi 2.6. My jsme pracovali s nejnovější verzi jádra.

V adresáři se zdrojovými kódy kernelu spustíme standardní konfiguraci, pro procesory OMAP, příkazem `armv7a-unknown-linux-gnueabi-make omap2plus_defconfig`. Posléze je nutné konfiguraci upravit, tedy spustíme textové konfigurační rozhraní `armv7a-unknown-linux-gnueabi-make menuconfig`.

Jednotlivé konfigurace jsou uloženy, i se zkompilovanými jádry, na přiloženém CD. Provedené úpravy jsou následující:

- přizpůsobení konfigurace
  - OMAP3530
  - deska BeagleBoard
  - odstranit nepotřebný zařízení, které se nevyskytuje na použitém kitu. Například síťová karta, ...
- uzpůsobení jádra pro RT práci
  - vypnout řízení spotřeby
  - vypnout řízení frekvence procesoru
  - vypnout co nejvíce ladicích systémů

Poslední příprava spočívá v nainstalování balíčku s pomocnými nástroji programu Das U-boot. Který se používá pro zavádění systému pro embedded zařízení a je použit i v této práci. Pro instalaci spustíme `emerge dev-embedded/u-boot-tools`. Pomocné nástroje jsou automaticky využity při kompilaci jádra. Samotnou kompilaci jádra spustíme příkazem `armv7a-unknown-linux-gnueabi-make uImage`. Volbou uImage říkáme kompilačnímu skriptu, že požadujeme jádro zkomprimované a připravené pro Das U-boot. Připravené jádro nacházíme v `./arc/arm/boot/uImage`.
Po ukončení kompilace je nutné spustit proceduru, která připraví adresář pro vytváření externích modulů. Tedy provedeme `armv7a-unknown-linux-gnueabi-make modules_prepare`. Přidání hlavičkových souborů, bez připravení pro přidávání externích modulů, způsobí chybu při kompilaci. Vytvoření vlastních modulů využijeme v další části práce.

3.1.3. Instalace systému Gentoo na architekturu armv7

Pro instalaci jsme vybrali systém Gentoo, který je možné předpřipravený stáhnout z distfiles.gentoo.org/releases/arm/autobuilds/current-stage3-armv7a. Aktuální repositář stáhneme z například gentoo.osuosl.org/snapshots/portage-latest.tar.xz.
Předpokládejme, že paměťová karta má připojené jednotlivé oddíly pod vytvořeními jmenovkami do adresáře `/media`. Pak můžeme rozbalit systém i repositář pomocí příkazů:

```
tar xjpf stage3-armv7a-20111220.tar.bz2 -C /media/gentoo/
tar xjpf portage-latest.tar.bz2 -C /media/portage/
```

V první části souboru `/media/gentoo/etc/fstab` definujeme přípojné body a jejich parametry. Pro vysvětlení, `/dev/mmcblk0p2` označuje druhý diskový oddíl na paměti v slotu pro karty. Pro náš příklad to bude:

```
/dev/mmcblk0p1  /boot           vfat   noauto,noatime  1 2
/dev/mmcblk0p2  /               ext3   noatime        0 1
/dev/mmcblk0p3  /usr/portage    ext3   noatime        0 1
```

Následuje vytvoření adresáře pro připojení repositáře a nastavení jeho správného vlastníka.

```
mkdir /media/gentoo/usr/portage
chown portage:portage /media/gentoo/usr/portage
```

Dále musíme nastavit heslo pro superuživatele, které můžeme vygenerovat pomocí `openssl passwd -1`. Výsledný řetězec vložíme do souboru `/media/gentoo/etc/shadow` na řádek, který začíná řetězcem „root“. A to tak, že jej zkopírujeme mezi první dvě dvojtečky na řádku. My jsme zvolili heslo welc0me.

Aby jsme se mohly připojit na sériovou konzoly přidáme na konec souboru `/media/gentoo/etc/securetty` tři řádky:

```
ttyO0
ttyO1
ttyO2
```

Jedná se o sériové porty, které se pod kitem BeagleBoard označují nestandardně. Zařízení ttyO2 je vyvedeno jako RS232 rozhraní na konektor IDC10. Proto následovně upravíme soubor `/media/gentoo/etc/inittab`:

```
# SERIAL CONSOLES
s0:12345:respawn:/sbin/agetty 115200 ttyO2 vt100
```

Důležité je nezapomenout změnit rychlost komunikace na 115200Bd.

Pro spuštění systému nám chybí už jen dvě věci. Již vzpomínaný program Das U-boot a binární soubor, který načte tento zavaděč systému. Následující příkazy vytvoří zavaděč systému:

```
git clone git://gitorious.org/x-loader/x-loader.git xloader
cd xloader
armv7a-unknown-linux-gnueabi-make omap3530beagle_config
armv7a-unknown-linux-gnueabi-make
```

Žádaný binární soubor zavaděče je pod názvem `MLO`, který, jako první, musíme zkopírovat na zaváděcí oddíl předpřipravené karty. Obdobně můžeme připravit program u-boot.

```
git clone git://arago-project.org/git/projects/u-boot-omap3.git
cd u-boot-omap3
armv7a-unknown-linux-gnueabi-make omap3_beagle_config
armv7a-unknown-linux-gnueabi-make
```

Kde je výsledným souborem je `u-boot.bin`, který zkopírujeme na paměťovou kartu pod názvem `u-boot`.

3.1.4. Nastavení jednotlivých pinů na rozšiřujícím rozhraní

U programu Das u-boot se na chvíli zastavíme. Poskytuje možnost nastavení módů jednotlivých pinů. Tyto je také možné měnit pod běžícím systémem, je-li v jádře zakompilována podpora „OMAP multiplexing support“ `CONFIG_OMAP_MUX` a zároveň „Multiplexing debug output“ `OMAP_MUX_DEBUG`.

V souboru `board/ti/beagle/beagle.h` nacházíme konfiguraci pro jednotlivé piny. Nastavení můžeme rozdělit do několika kroků, které si uvedeme na příkladě pro UART2_TX:

1. V tabulce 7-4 na straně 774 v manuálu [11] najdeme chtěný signál, jak je popsáno v části 2.2. Pro náš příklad jde o UART2_TX.
2. V hlavičkovém souboru najdeme příslušný registr (MCBSP3_CLKX), který je zároveň vyveden na konektor (pin 6), a nastavíme ho do žádaného módu, tedy módu jedna.
3. Musíme správně nastavit povahu pinu, vstupní nebo výstupní.
4. Nastavit pull-up, pull-down rezistor. V příkladě použití sériového rozhraní není potřeba jeho funkčnosti, tedy ho deaktivujeme.
5. Abychom předešli zdvojenému namapování pinů, všechny ostatní možnosti zapnutí funkce UART2_TX přepneme do jiného módu.

Celý příklad bude vypadat následovně:

```
/*
 * IEN  - Input Enable
 * IDIS - Input Disable
 * PTD  - Pull type Down
 * PTU  - Pull type Up
 * DIS  - Pull type selection is inactive
 * EN   - Pull type selection is active
 * M1   - Mode 1
 * The commented string gives the final mux configuration
 */

MUX_VAL(CP(MCBSP3_CLKX),     (IDIS | PTD | DIS | M1)) /*UART2_TX*/\
MUX_VAL(CP(UART2_TX),        (IEN  | PTD | DIS | M4)) /*GPIO_146*/\
```

Po zorientování v hlavičkovém souboru jsme zjistili, že sériové rozhraní UART2 je standardně zapnuto na pinech čtyři a šest rozšiřujícího slotu. Pro ověření provedeme na běžícím systému následující příkazy:

```
cat /sys/kernel/debug/omap_mux/uart2_tx
name: uart2_tx.gpio_146 (0x48002178/0x148 = 0x0104), b aa25, t NA
mode: OMAP_PIN_INPUT | OMAP_MUX_MODE4

cat /sys/kernel/debug/omap_mux/cat mcbsp3_clkx
name: mcbsp3_clkx.uart2_tx (0x48002170/0x140 = 0x0001), b af5, t NA
mode: OMAP_PIN_OUTPUT | OMAP_MUX_MODE1
```

V krátkosti můžeme říct, že do daných souborů se dá zapisovat i hexadecimální číslo a tím nastavoval povahu pinu.

3.2. Vývoj pro architekturu armv7

S již připraveným nástrojem toolchain pro architekturu armv7, můžeme začít kompilovat aplikace pro BeagleBoard na hostujícím počítači. Což jsme vlastně již udělali, při tvorbě jádra a zaváděcích nástrojů. Stačí před standardní nástroje, jako `gcc, cc, make, strip`, přidat `armv7a-unknown-linux-gnueabi-`. Následující nastavení překladače plně využívá výhod procesoru OMAP3530

```
-O2 -march=armv7-a -mcpu=cortex-a8 -mfpu=neon -ftree-vectorize
-mfloat-abi=softfp -ffast-math -fsingle-precision-constant -g3
```

Nastavení bylo odzkoušeno, ale v samotných testech jsme se snažili o co nejvíce atomický průběh programu. Tedy bez optimalizace, která snižuje determinovanost chování aplikace.

3.2.1. Práce s vývojovým nástrojem Eclipse

Při práci bylo použito vývojové prostředí Eclipse. Které jsme nainstalovali následovně:

```
layman -a seden
emerge dev-util/eclipse-sdk
```

První z příkazů přidává externí repositář, kde je obsažena novější verze programu Eclipse-3.7.1-r7, která je použita v práci. Instalaci můžeme provést i podle manuálu ze stránek projektu wiki.eclipse.org. Do samotného prostředí jsme doinstalovali CDT (modul pro vývoj v C/C++), přes menu Help - Install New Software.
Není-li uvedeno jinak, platí pro všechny projekty následující:

- jedná se o projekty v jazyce C se standardem z roku 1999 (-std=c99)
- je zapnuta maximální možná úroveň hlášení varování (-pedantic -pedantic-errors -Wall -Wextra)
- v možnostech je zvolen toolchain „Linux gcc“
- je využívána možnost automatické tvorby makefile, souboru definující postup sestrojení projektu pomocí nástroje make, který je uložen v složce Debug
- sestavovacímu programu není předán parametr -static, tedy spustitelný soubor neobsahuje všechny použité knihovny
- sestavovacímu programu je předán parametr -lrt. Jedná se o přidání knihovny POSIX.1b Realtime Extensions library, potřebné pro práci s časovačem a vlákny

Každý vytvořený projekt v práci je možné nalézt na přiloženém CD.

3.2.2. První aplikace typu Hello ARM

Pomocí menu File - New - C Project, se necháme provést vytvořením nového projektu. Ve vlastnostech projektu v části C/C++ Buil - Settings - Tool Settings následovně zaměníme spouštěné příkazy pro kompilátor, linker a assembler.

GCC C Compiler
: gcc nahradíme příkazem armv7a-unknown-linux-gnueabi-gcc

GCC C Linker
: gcc nahradíme příkazem armv7a-unknown-linux-gnueabi-gcc

GCC C Assembler
: as nahradíme příkazem armv7a-unknown-linux-gnueabi-as

Dále můžeme v menu nastavit parametry pro gcc uvedené na začátku kapitoly 3.2.
Samotný zdrojový kód může být následující:

```
#include <stdio.h>
#include <stdlib.h>

int main (void)
{

        printf("Hello␣ARM\n");
        return 0;
}
```

Po zkompilování není výsledný soubor spustitelný na hostitelské architektuře x86_64. Ale po spuštění na BeagleBoard pod Linuxem se nám vypíše řetězec „Hello ARM“.

V samotné práci byla snaha o co největší přenositelnost, tedy zdrojové kódy jsme kompilovali pod oběma architekturami samostatně. Všechny výsledné aplikace jsou odladěné pro obě architektury. Při psaní zdrojového textu jsme nepoužívali funkce, které obsahovali v názvu _np_ (non portable - nepřenositelné). Až při tvorbě jaderných modulů byl využit toolchain pro armv7. Kdy jsme na BeagleBoard přenášeli již hotové moduly pro bežící jádro.

4. Způsob analýzy RTOS

Analýzu RTOS můžeme provést několika způsoby:

- Analýza kódu
  Procedura vyžaduje zdrojové kódy a podrobnou znalost použitého procesoru. Považuje se za nákladnější než ostatní možnosti.
- Měření zpoždění
  - Hardwarové
    Sledujeme reálný signál, při použití logických analyzátorů dosahujeme nejvyšších přesností.
  - Softwarové
    * Podpora měření latencí přímo v jádru
    * Balíček programů rt-test
    * Vlastní aplikací

4.1. Možné způsoby kvalifikace RTOS

Zde je uvedeno několik možných softwarových analýz RTOS:

- **Rychlost přepínání vláken**
  Několik vláken soupeří o zdroje,které si lineárně rozdělují, jak uvádí [13]. Každé vlákno si po získání a uvolnění zdroje inkrementuje svůj vlastní čítač. Na konci testu bude kvalifikačním parametrem součet čítačů všech vláken.
- **Rychlost zpracování požadavku přerušení**
  Jedno vlákno probouzí druhé pomocí softwarového přerušení. To signalizuje prvnímu vláknu, že je probuzeno a ihned se uspí. První vlákno inkrementuje proměnou. Její hodnota, za měřený časový úsek, je kvalifikačním parametrem.
- **Test meziprocesní komunikace**
  Vlákno zapisuje a posléze čte zprávu, z fronty zpráv, v nekonečném cyklu. Po každém proběhnutí cyklu je inkrementována proměnná. Její hodnota, za měřený časový úsek, je kvalifikačním parametrem.
- **Test synchronizačních primitiv**
  Vlákno samo uvolňuje a uzamyká jedno ze synchronizačních primitiv v nekonečném cyklu. Po každém proběhnutí cyklu je inkrementována proměnná. Její hodnota, za měřený časový úsek, je kvalifikačním parametrem.
- **Měření nepřesnosti časovače**
  Vlákno se uspí na předem známý časový úsek. Před a po uspání zjistí čas systémovým voláním. Rozdíl mezi očekávanou dobou uspání a dobou naměřenou je výsledkem měření. Příkladem může být známý program `cyclictest`.

4.2. Implementované testy

V samotné implementaci jsme používali časovače vestavěné v jádře, pro zaručení přenositelnosti. Pro tyto funkce jádro automaticky používá časovač s vysokým rozlišením. Čas v jádře můžeme rozdělit na dvě základní skupiny: reálný čas a monotonní čas. Reálný čas je běžný čas, který se počítá v sekundách od počátku roku 1970. Toto datum někdy nazýváme počátkem Unixového věku. Jádro nastavuje reálný čas při startu a to pomocí volání hardwarových hodin. Při používání tohoto času nesmíme zapomínat, že jej lze seřizovat. Například deamon NTP (Network Time Protocol) průběžně modifikuje rychlost běhu tohoto času, ve snaze nastavit přesný čas. A to může být nežádoucí při měření časových intervalů.

S monotonním časem nelze manipulovat a je odvozen od taktu procesoru. Počítadlo tohoto času začíná běžet při startu systému a po celou dobu běhu běží stejnoměrně. Je méně přesný než čas reálný, ale libovolné relativní časy se počítají správně. Tento druh časovače je doporučeno používat u periodických úloh, kde je kladen důraz na přesnost periody. V práci jsme použili právě tento druh časovače. Časovač má nanosekundovou rozlišitelnost, ale jak uvádí Stewart [18] jeho přesnost dosahuje jednotky mikrosekund.

Při tvorbě testů jsme použili vlákna, proto si v rychlosti vysvětlíme v práci použité funkce. Podrobný popis jednotlivých funkcí najdeme na manuálových stránkách [15].

- `pthread_create`
  Funkce vytvoří vlákno. Pro nás je důležitá možnost, právě vytvářenému vláknu, určit vstupním parametrem jeho prioritu a politiku plánovače. A to druhým parametrem, který je typu `pthread_attr_t`.

- `pthread_join`
  Tuto funkci využijeme při čekání na ukončení vlákna. Ale ona sama vlákno neukončuje, ani neprovádí úklid na-alokovaných prostředků.

- `pthread_close`
  Funkce okamžitě ukončí vlákno a uvolní všechny prostředky náležící vláknu.

- `pthread_attr_setschedpolicy`
  Pomocí této funkce je možné nastavit politiku plánování pro vlákno. Nastavení se zapisuje do proměnné typu `pthread_attr_t`.

- `pthread_attr_setschedparam`
  S touto funkcí určíme prioritu procesu, kterou funkce zapíše do proměnné typu `pthread_attr_t`. Samotná hodnota priority je uložena v struktuře `sched_param`.

Jelikož se jedná o triviální případ výměny dat mezi vlákny, řešili jsme ho vytvořením velice jednoduchého lineárního seznamu. Kde se příchozí prvek se vždy ukládá na konec seznamu. Funkce pro čtení odebírá prvek ze začátku seznamu. Jedná se tedy o FIFO zásobník. Při vytváření nového záznamu používáme funkci `malloc`, která při svém vykonávaní blokuje všechna systémová volání. Co je z hlediska realtime aplikace nevhodné.

Pro implementaci jsme vybrali tyto testy:

1. **Měření chvění časovače**
   Program testuje přesnost časovače. Porovnává teoretickou dobu uspání vlákna s reálnou.

2. **Měření pomocí sériového rozhraní**
   Program měří čas přenesení dat přes sériovou linku.

3. **Přerušení externím signálem**
   Na vstupní pin je přiváděn periodický obdélníkový signál. Obslužná rutina IRQ informuje o příchozím přerušení pomocí GPIO.

4. **Periodická úloha**
   Test vytvoří časovače pro periodické úlohy. Úlohy vytvářejí signál na GPIO výstupu. Měřením je ověřena spolehlivost časovačů.

5. **Generace PWM signálu na GPIO**
   Byl použit projekt pro generaci PWM signálu. U kterého jsme si ověřili přesnost a stabilitu generovaného signálu.

Výstupy jsou přímo ukládány do souboru, který vzniká na základě časové značky v pracovním adresáři. V následujících kapitolách je funkce jednotlivých programů podrobněji rozepsána. Samotné zdrojové kódy jsou na přiloženém CD.

user space		kernel space		
`gettimeofday`	`clock_gettime`	`getnstimeofday`	`getrawmonotonic`	`do_gettimeofday`
rozlišení v μs	rozlišení v ns	rozlišení v ns	rozlišení v ns	rozlišení v μs

Tabulka 4.1: Přehled použitých funkcí pro získání času

4.2.1. Měření chvění časovače

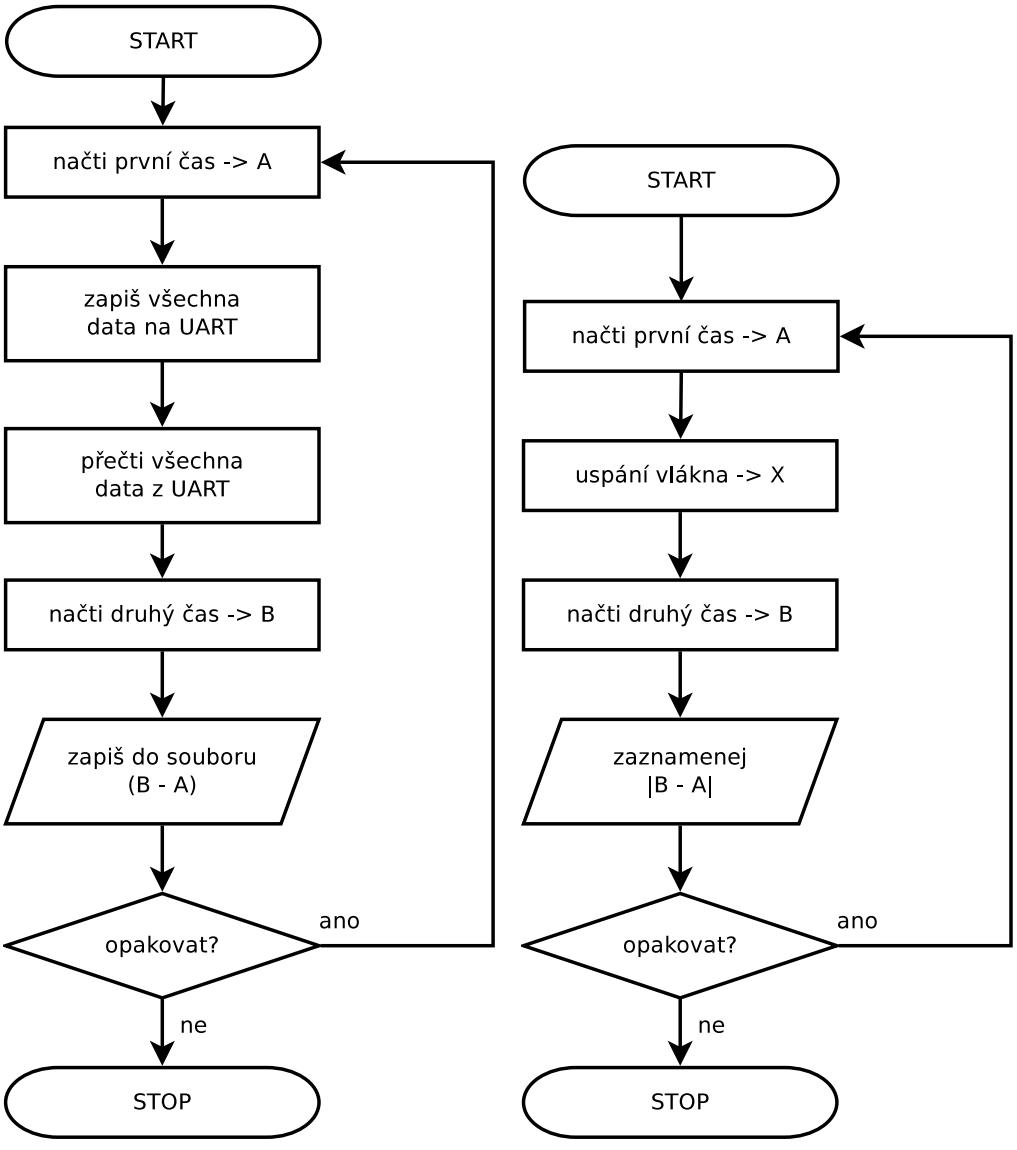
Program akceptuje dva vstupní parametry: počet opakování měření a délku uspání. Vývojový diagram průběhu měření znázorňuje obrázek 4.1b.
Měřící vlákno se uspí na stanovenou dobu. Výsledkem je rozdíl času před uspání a času získaném po uspání. test byl proveden dvěma způsoby. V prvním zjišťování času provádíme pomocí `clock_gettime`. Která je doporučována, využívá časovače s vysokým rozlišením, z podporovaných funkcí 4.1. Druhou možností je přečtení registru CCNT (Cycle Counter Register). Který se inkrementuje při každém cyklu. Při známé frekvenci jádra 600 MHz, každý cyklus reprezentuje 1.67 ns. Čtení z registru je nutné povolit v privilegovaném režimu, tedy modulem zavedeným v jádře.

Program neověřuje práva pro spouštění procesů s RT prioritou. Proto je v začátku implementovaný jednoduchý test, který zjišťuje, jestli je spuštěn pod super uživatelem. Posléze nastaví pro běžící proces plánovač Round-Robin s maximální prioritou. Samotná naměřená data se zaznamenávají do předem alokovaného pole. Tedy je potřeba počítat s konečnou velikostí paměti RAM při volbě počtu měření. Zápis dat do souboru se provádí až po samotném měření. Pro minimalizaci ovlivnění naměřených dat.

4.2.2. Měření pomocí sériového rozhraní

Vstupní parametry pro test jsou počet opakování, délka zasílaného řetězce a sériové zařízení. Aplikace náhodně vygeneruje řetězec o požadované délce. Ten pak odešle na sériový port a čeká na odpověď. Příchozí řetězec otestuje, zda se shoduje se zaslaným. Je-li shodný uloží dobu trvání mezi odesláním a přijmutím řetězce. Tento postup znázorňuje diagram 4.1a.

Měření se spouští s vláknem s normální prioritou. A poté s vláknem s maximální prioritou.

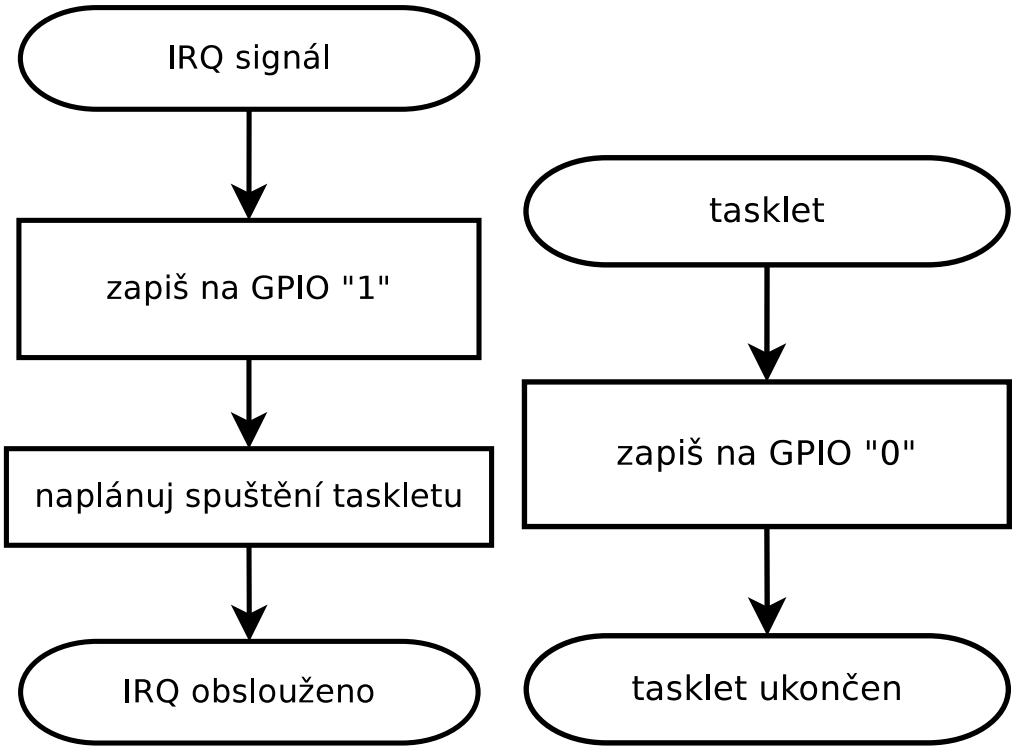


(a) Měření zpoždění na sériovém portu (b) Měření časového chvění

Obrázek 4.1: Zjednodušené vývojové diagramy

4.2.3. Přerušení externím signálem

Registraci obslužné rutiny pro přerušení je možné provést jen v privilegovaném režimu. Proto je nutné použít jaderný modul. Oba testy reagují na náběžnou hranu signálu na pinu 4, tedy GPIO 139. Generovaný signál musí ležet v rozmezí nula až 1,8 V. V opačném případě hrozí poškození vstupů. Test při detekovaném přerušení zapíše na výstupní pin 5, GPIO 138, logickou hodnotu jedna. Posléze naplánuje tasklet, který na výstup zapíše logickou nulu, viz. vývojový diagram 4.2a a 4.2b.
Náběžná hrana signálu na výstupném pinu bude posunuta v časové ose vůči generovanému signálu. Tento posun si můžeme zobrazit na osciloskopu, nebo přímo měřit čítačem. Tato doba reprezentuje IRQ zpoždění.



(a) Implementace přerušení na GPIO (b) Tasklet pro zapsaní "0"na výstup

Obrázek 4.2: Zjednodušené vývojové diagramy

4.2.4. Periodická úloha

V kernel space jsou vytvořeny dva periodické časovače se stejnou periodou, ale s posunutým startem o polovinu periody. První z nich nastavuje na výstupní GPIO pin logickou jedničku. Druhý časovač na výstupní pin nastavuje logickou nulu. Na výstupu by měl vzniknout obdélníkový signál s definovanou periodou a střídou 50 procent.

4.2.5. Generace PWM signálu na GPIO

Byl použit projekt servodrive ze stránek https://github.com/tallakt/servodrive.

4.2.6. Implementace zátěže

V těle programu se spouští následující vlákna:

- **Dynamická alokace paměti**
  Vlákno v nekonečné smyčce dynamicky alokuje paměť, kterou vyplní znakem Q. Alokace paměti probíhá pomocí funkce `malloc`. Po alokaci jednoho kilobytu se první byt uvolní voláním `free` a následně se znovu alokuje.

- **Generace IRQ**
  Zde je použito volání funkce `gettimeofday`, která musí pro zjištění aktuálního času vygenerovat několik přerušení.

- **Generace I/O zátěže**
  V nekonečné smyčce se provádí funkce `sync`, ta synchronizuje paměť cache pro souborové systémy.

- **Generace nových procesů**
  Zde rodičovský proces vytváří potomka, který po vytvoření ukončí svoji činnost. Původní proces čeká na ukončení potomka a vzápětí vytváří nového potomka.

- **Čtení dat z paměti**
  Prvním krokem je alokace, a vyplnění ‚paměti o velkosti dvojnásobku součtu cache pamětí procesoru. To nám zaručuje, že procesor bude nucen pracovat s pamětí RAM.

- **Vytvoření CPU zátěže**
  Ve vláknu se v nekonečné smyčce počítá druhá odmocnina z náhodných čísel.

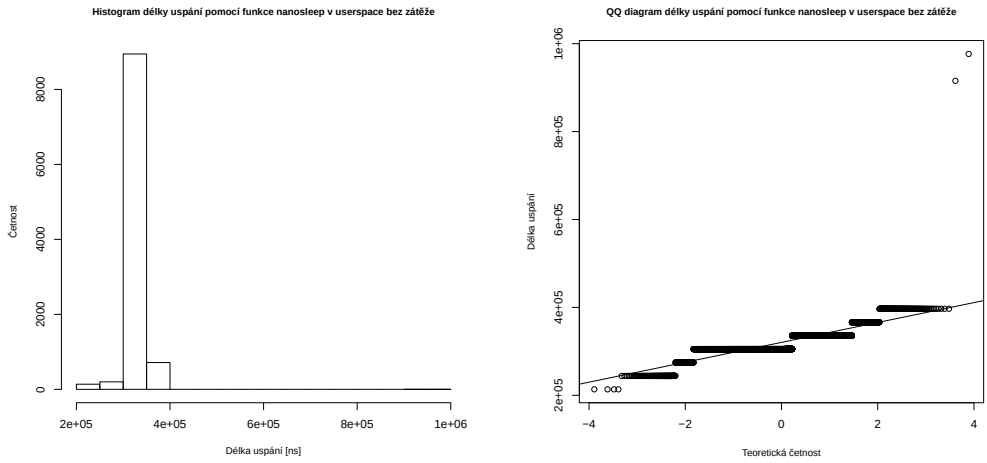
V samotném testu byla variabilita zátěže vytvořena zakomentováním volání, které vytvářely patřičné vlákno se zátěží.

5. Vyhodnocení naměřených dat

Testy byly spouštěny pomocí skriptů, která jsou obsahem přiloženého CD. Statistickou analýzu pomocí experimentu jsme prováděli dle [5].

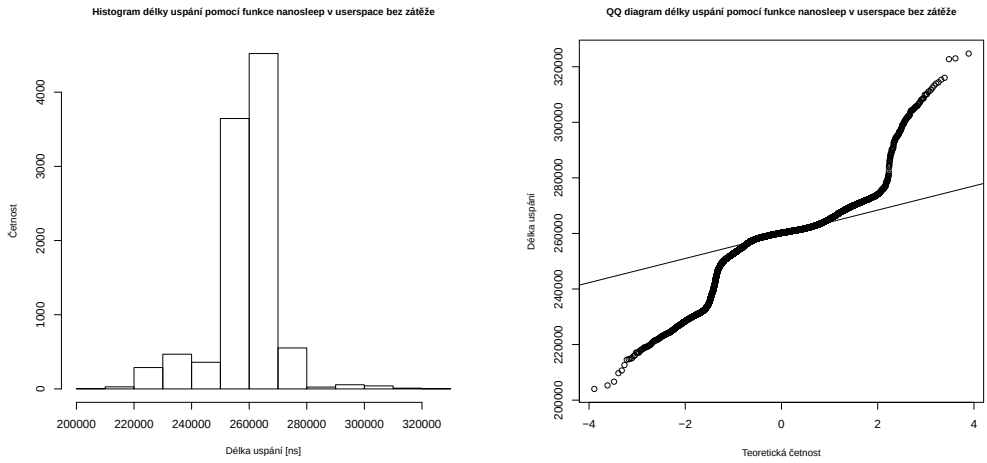
5.1. Měření chvění časovače

První analýzu dat jsme provedli na vzorku 10 000 měření s dobou uspání 100 μs. Výsledný histogram a kvantil-kvantil graf pro měření pomocí funkce `clock_gettime` je zobrazen na obrázku 5.1. Volání pro zjištění rozlišitelnosti časovače vrací hodnotu 1 ns, ale výsledné grafy vykazují značně diskrétní rozdělení výsledných hodnot.



(a) Histogram naměřeních dat (b) Kvantil-kvantil graf naměřeních dat

Obrázek 5.1: Analýza naměřených dat pomocí pomocí prvního testu



(a) Histogram naměřeních dat (b) Kvantil-kvantil graf naměřeních dat

Obrázek 5.2: Analýza naměřených dat pomocí pomocí prvního testu

Další měření jsme provedli pomocí odečítání CCNT registru za stejných podmínek. Výsledné grafy jsou zobrazeny na obrázku 5.2. Hodnoty již vykazují spojité rozložení. Snížení průměrné naměřené hodnoty z 320 μs na 260 μs je zapříčiněno rychlostí vykonání procedury pro získání času. Normalitu rozdělení ověříme pomocí t-testu, Pearson chi-square testu, Anderson-Darling testu a Kolmogorov-Smirnov testu, viz. výstup z programu R:

```
> t.test (ND_clock)

        One Sample t-test

data:  ND_clock
t = 1265.742, df = 9999, p-value < 2.2e-16
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
 318752.3 319741.1
sample estimates:
mean of x
 319246.7

> pearson.test (ND_clock, adjust = TRUE)

        Pearson chi-square normality test

data:  ND_clock
P = 331191.6, p-value < 2.2e-16

> ad.test (ND_clock)

        Anderson-Darling normality test

data:  ND_clock
A = Inf, p-value = NA

> lillie.test (ND_clock)

        Lilliefors (Kolmogorov-Smirnov) normality test

data:  ND_clock
D = 0.2972, p-value < 2.2e-16
```

Všechny testy potvrzují normalitu dat na hladině významnosti pěti procent. Dalším krokem je příprava třídícího experimentu pro šest faktorů zátěže o dvou hladinách. Zde také využijeme R. Zvolili jsme design o dvanácti bězích. Vygenerovaný experiment, i s naměřenými daty v ns, má následující podobu :

	malloc_memory	IRQ	io	fork	read_memory	cpu_burn	mean	std dev
1	ON	OFF	OFF	OFF	ON	OFF	257374.7	15138.9
2	OFF	OFF	OFF	ON	OFF	ON	258761.7	38736.96
3	ON	OFF	ON	ON	OFF	ON	258672.1	40279.22
4	ON	ON	OFF	ON	ON	ON	258281.2	22259.02
5	OFF	ON	OFF	ON	ON	OFF	258683.9	27584.57
6	OFF	ON	ON	OFF	ON	ON	257535.7	16087.56
7	ON	OFF	ON	ON	ON	OFF	258535.5	27410.11
8	OFF	OFF	OFF	OFF	OFF	OFF	324905.1	31260.93
9	ON	ON	ON	OFF	OFF	OFF	258147.8	17771.62
10	ON	ON	OFF	OFF	OFF	ON	257861.3	9558.313
11	OFF	ON	ON	ON	OFF	OFF	258406.4	11542.43
12	OFF	OFF	ON	OFF	ON	ON	258424.0	13311.37

V návrhu se vyskytují i další faktory e1, e2, e3, e4 a e5. Jedná se o nepravé faktory, které zde figurují pro správný výpočet. Jako responsní veličinu jsme zvolili průměr a směrodatnou odchylku ze 100 000 naměřených dat pro každý běh. Délka uspání byla ponechána na 100 μs. Výsledné vliv faktorů je zobrazen v příloze na obrázcích C.1, C.2, C.3, C.4. Žádný faktor, ani na hladině významnosti deset procent, významně neovlivňuje průměrnou hodnotu délky uspání. Tedy vykonávání úlohy nebylo ovlivněno zátěží, což je předpoklad pro RT systému. Zajímavým výsledkem je snížení průměrné délky uspání při zapnutí jakékoliv zátěže. Tento jev může být způsoben snížením časového kvanta přidělovaného plánovačem.

Pro směrodatnou odchylku je z grafu C.3 předpoklad, že syntetická zátěž v podobě emulace vytváření procesů negativně ovlivňuje směrodatnou odchylku. Odhad efektů faktorů pro lineární aproximaci vlivu, pomocí programu R, je:

```
formula = std.dev ~ malloc_memory + IRQ + io + fork +
    cpu_burn + red_memory, data = nanosleep.DoE

Coefficients:
                Estimate Std. Error t value Pr(>|t|)
(Intercept)      22578.4     2687.9   8.400 0.000392
malloc_memory1    -508.9     2687.9  -0.189 0.857284
IRQ1             -5111.2     2687.9  -1.902 0.115634
io1              -1511.4     2687.9  -0.562 0.598195
fork1             5390.3     2687.9   2.005 0.101240
cpu_burn1          793.7     2687.9   0.295 0.779656
red_memory1      -2279.8     2687.9  -0.848 0.435050

Residual standard error: 9311 on 5 degrees of freedom
Multiple R-squared: 0.6376,     Adjusted R-squared: 0.2027
F-statistic: 1.466 on 6 and 5 DF,  p-value: 0.3457
```

Model nesplňuje kritérium statistické významnosti. Tedy nezle z něho vyvozovat závěry.

5.2. Měření pomocí sériového rozhraní

V tomto scénáři jsme propojili vysílací a přijímací vodiče na sériovém výstupu. Cokoliv zapíšeme, po této úpravě, na sériové rozhraní se beze změny přijme na stejném rozhraní. Pro UART3, IDC-10 konektor, jde o piny dva a tři. UART2 je vyveden na rozšiřujícím slotu a vysílací linka je na pinu šest a přijímací na pinu osm.

První test byl proveden na rozhraní ttyO1, které se v dokumentaci označuje UART2. Existující OMAP sériové rozhraní vypíše následující příkaz:
`cat /proc/tty/driver/OMAP-SERIAL`. Po spuštění program fungoval podle předpokladů a testu z architektury x86. Avšak po několika tisících měřících cyklech program neregulérně ukončil svoji činnost. Po mnohanásobném odzkoušení program ani jednou neukončil svoji činnost regulérně. V některých případech došlo k nefunkčnosti celého systému. Tato vlastnost byla nezávislá na použitém jádru. V samotném výpisu jádra nebylo nalezeno nic korespondující s popsaným problémem.

Další test byl proveden na rozhraní ttyO2. Jelikož pomocí tohoto zařízení bylo navázáno terminálové spojení, byl poupraven spouštěcí skript. Bylo zde přidáno desetisekundové čekání před zahájením ostatních činností. V průběhu tohoto čekání byl vyměněn sériový konektor DE-9 za konektor propojující vysílací a přijímací kanál. V průběhu tohoto testu nebyly zaznamenány žádné problémy. Souhrnné výsledky pro posílání řetězce o délce 25 znaků, jsou prezentovány v tabulkách 5.1, 5.2 a 5.3. Jádro bez podpory preempce má uvedené jen hodnoty bez spuštěné zátěže. Jelikož po spuštění simulace zátěže přestalo reagovat.

Je patrné, že jádro s podporou RT výrazně snížilo maximální časovou prodlevu. Nečekaným výsledkem jsou rychlé odezvy v běžné plánovací politice, v testu, kdy byl OS v zátěži. Jedním z vysvětlení je možnost nesprávné funkčnosti časovače v době zátěže. Která by se dala ověřit implementací počítaní časových rozdílů na bázi tiků mikroprocesoru.

	SCHED_OTHER	**SCHED_FIFO**
minimum	30.8 ms	30.9 ms
medián	30.9 ms	30.7 ms
maximum	1.00 s	31.2 ms

Tabulka 5.1: Jádro bez podpory preempce

	SCHED_OTHER		**SCHED_FIFO**	
	bez zátěže	se zátěží	bez zátěže	se zátěží
minimum	30.9 ms	14.1 ms	30.9 ms	30.9 ms
medián	31.0 ms	15.6 ms	31.2 ms	31.2 ms
maximum	1.00 s	16.4 ms	978 ms	33.4 ms

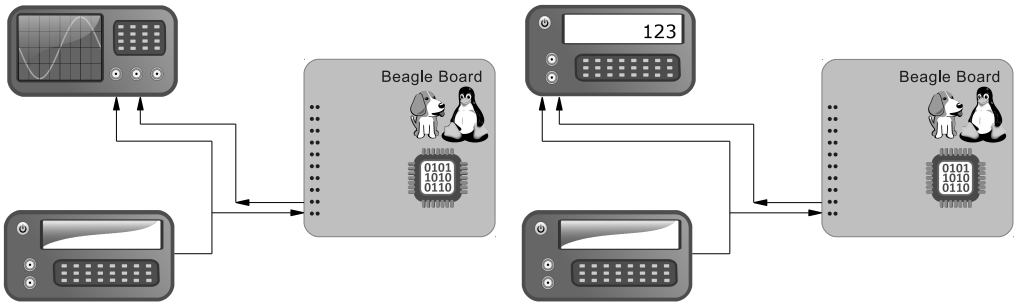
Tabulka 5.2: Jádro s podporou preempce

	SCHED_OTHER		**SCHED_FIFO**	
	bez zátěže	se zátěží	bez zátěže	se zátěží
minimum	31.1 ms	931 μs	31.1 ms	31.0 ms
medián	31.2 ms	15.6 ms	31.2 ms	31.2 ms
maximum	37.6 ms	23.6 ms	43.2 ms	42.6 ms

Tabulka 5.3: Výsledky pro RT jádro

5.3. Přerušení externím signálem

Pro další testy jsme použili generátor signálu. Kde byla zvolena velikost napětí špička-špička 1.6 V, ofset 0.8 V, obdélníkový tvar se střídou 50 %.



(a) Měření pomocí osciloskopu (b) Měření pomocí čítače

Obrázek 5.3: Zapojení pro měření latence přerušení

První test jsme provedli s osciloskopem 5.3a. Při frekvenci 1 kHz byl hrubý odečet zpoždění od 31,5 μs do 36,6 μs. Šířka pulsu, doba než proběhne tasklet, byla 9 μs. Dalším krokem bylo zvyšování frekvence. Systém začal vynechávat zpracování příchozích přerušení na frekvenci 8,366 kHz. Převrácenou hodnotou dostáváme 120 μs. Tato hodnota je pro systém významná, jelikož rychlejší děje není schopen zpracovávat.

Další měření probíhalo pomocí čítače 5.3b. Prvně jsme zkusili odhadnout délku provedení funkce pro změnu hodnoty na GPIO ze vzorku 2 500 hodnot pro každé měření. Průměrná délka latence při použitý této funkce je 32,5 μs. Když voláme tuto funkci dvakrát po sobě dostáváme 36.3 μs. První volaní nastaví na výstup logickou nulu a až druhé volání logickou jedničku. Při přímém zápisu do adresního prostoru na GPIO naměříme latenci 28,9 μs Přibližná délka provedení volání `gpio_set_value` je 3,8 μs. Zápisem přímo do adresního prostoru provedeme stejný úkon o 3,6 μs rychleji.

Pro analýzu DoE jsme zvolili blok o velikosti 10 000 vzorků. Z kterého jsme posuzovali průměrnou hodnotu a směrodatnou odchylku. Tyto dvě hodnoty nám poskytl čítač se statistickými funkcemi. Design a zátěže jsme použili z kapitoly 5.1. Výsledné hodnoty jsou:

	malloc_memory	IRQ	io	fork	read_memory	cpu_burn	mean	std dev
1	ON	OFF	OFF	OFF	ON	OFF	19.411	2.513
2	OFF	OFF	OFF	ON	OFF	ON	29.203	7.497
3	ON	OFF	ON	ON	OFF	ON	29.552	3.273
4	ON	ON	OFF	ON	ON	ON	33.082	24.415
5	OFF	ON	OFF	ON	ON	OFF	29.167	3.273
6	OFF	ON	ON	OFF	ON	ON	24.971	4.400
7	ON	OFF	ON	ON	ON	OFF	29.287	5.866
8	OFF	OFF	OFF	OFF	OFF	OFF	19.647	3.513
9	ON	ON	ON	OFF	OFF	OFF	23.771	4.409
10	ON	ON	OFF	OFF	OFF	ON	22.530	3.572
11	OFF	ON	ON	ON	OFF	OFF	28.978	6.485
12	OFF	OFF	ON	OFF	ON	ON	19.618	2.184

Rovněž jsme sledovali maximální dobu latence, kterou jsme nezahrnovali do experimentu pro její nenormální rozdělení. Maximální zpoždění přes všech dvanáct běhů bylo 210,591 μs. Výsledné grafy závislosti faktorů jsou zobrazeny v příloze C.5, C.6, C.7, C.8. Pro směrodatnou odchylku není ani jeden faktor významný. Průměrná hodnota je ovlivněna zátěží IRQ a fork. Odhad efektů faktorů pro lineární aproximaci vlivu to potvrzuje:

```
formula = mean ~ malloc_memory + IRQ + io + fork +
    cpu_burn + red_memory, data = gpio_irq_DoE

Coefficients:
               Estimate Std. Error t value Pr(>|t|)
(Intercept)     25.7681     0.4231  60.909 2.26e-08
malloc_memory1   0.5041     0.4231   1.192 0.286915
IRQ1             1.3151     0.4231   3.109 0.026595
io1              0.2614     0.4231   0.618 0.563680
fork1            4.1101     0.4231   9.715 0.000196
cpu_burn1        0.7246     0.4231   1.713 0.147436
red_memory1      0.1546     0.4231   0.365 0.729763

Residual standard error: 1.466 on 5 degrees of freedom
Multiple R-squared: 0.9561,     Adjusted R-squared: 0.9034
F-statistic: 18.15 on 6 and 5 DF,  p-value: 0.002982
```

Model splňuje statisticky významnou hladinu pět procent. Nejvyšší odhad efektu má zátěž fork. Je skoro čtyřnásobně vyšší než odhad efektu pro zátěž IRQ. Můžeme usuzovat, že je nevhodné generovat veliké množství nových procesů při běhu RT aplikací.

5.4. Periodická úloha

Měření probíhalo pomocí čítače. Výsledky jsme odečítali ze statistiky zabudované v čítači. Zde jsme zvolili 10 000 hodnot pro výpočet statistiky. Výsledné hodnoty jsou zaneseny v tabulce 5.4. Časovače vykazují systematickou chybu a rovněž vysokou variabilitu. Není je možné doporučit pro generaci přesně tvarovaných signálů.

nastavena frekvence	1000 Hz	10 000 Hz
změřená frekvence	1184,6 Hz	13 587 Hz
maximální odchylka střídy	22.3 %	25.2 %

Tabulka 5.4: Výsledky pro periodický signál

5.5. Generace PWM signálu

Po spuštění modulu jsme zobrazili výstupní signál na osciloskopu. Nastavené a změřené šířky pulzů jsou v tabulce 5.5. Změřené hodnoty neodpovídají nastavením a taky lze z předchozí kapitoly předpokládat nestabilitu střídy.

nastavena hodnota	1100 μs	1500 μs	1900 μs
naměřena hodnota	832 μs	1152 μs	1464 μs

Tabulka 5.5: Výsledky pro šířky pulsů generovaných pomocí PWM modulu

6. Závěr

V průběhu práce jsme se seznámili s kitem BeagleBoard a s Linuxovím prostředím. Jádrem samotného vývojového kitu je výkonný procesor ARM, který umožňuje běh standardního operačního systému. V průběhu práce vytvoříme cross kompilátor. Pomocí kterého vytvoříme zavaděč systému a samotné jádro Linuxu, na které je aplikováno realitme rozšíření.

První test odhalil nespojitost v získaných datech při použití volání pro získávání času. Dále jsme implementovali měření časového intervalu pomocí odečtení registru CCNT. Data naměřené touto metodou vykazovali spojitost a normalitu. Proto jsme mohli navrhnout a provést experiment. Po analýze naměřených dat se ani jeden odhad efektu faktoru námi navrhnuté syntetické zátěže neukázal jako statisticky významný. A to ani pro průměrnou hodnotu, ani pro směrodatnou odchylku, naměřených hodnot. To plně odpovídá stabilitě RT systému, tedy RT úloha není ovlivněna zátěží. Překvapením bylo kratší uspání při běhu zátěže. Které připisujeme kratším časovým kvantům přidělovaným plánovačem.

Test pomocí sériového rozhraní ukázal značné snížení maximální doby přenosu dat při použití RT patch. Což v praxi, například při komunikaci s měřícím zařízením pomocí SPI, znamená důležitou vlastnost. Rovněž můžeme říct, že nenastalo výrazné zhoršení průměrné doby zaslání dat po sériovém rozhraní.

Při měření latence přerušení na GPIO výstupech jsme v prvním kroku zkoušeli vhodnost implementace procedury `gpio_set_value` v jádře. Ukázala se jako nevhodná. Pomocí zapisování bitu přímo do adresního prostoru, je možné zrychlit odezvu o 3,6 μs. Což přibližně deset procent z celkového zpoždění. Dalším krokem bylo provedení experimentu se zátěží. Statisticky významným faktorem byla zátěž pomocí vytváření nových procesů a také IRQ zátěž pomocí volání `gettimeofday`. Lze doporučit nevytvářet mnoho nových procesů při běhu RT úlohy. Literatura striktně nedoporučuje používat volání `malloc`, jelikož obsahuje nepřerušitelné části. V experimentu se tento faktor neprojevil jako významný.

Časování periodických úloh pomocí časovače s vysokým rozlišením je zatíženo systematickou chybou a značnou variabilitou. Což se také ukázalo při testu projektu pro generaci PWM signálu pomocí výstupu na GPIO. Toto řešení není možné doporučit.

Dále práce může pokračovat v provedení syntetických testů o více faktorech. Jednotlivé faktory atomizovat na API volání v kernel space spuštěné v modulu jádra. Nevýhodou je množství volání, tedy i množství faktorů pro test. Zajímavým řešením by byla analýza běžně používaných aplikací. A následná impementace zátěže na nejčastěji používané volání.

Literatura

[1] BOVET, Daniel P., CESATI Marco: *Understanding the Linux Kernel.* [kniha] Boston, MA: Safari Tech Books Online, 2005. 944 s. ISBN 978-0596005658.

[2] *Cortex-R Series - ARM* [on-line] www.arm.com/products/processors/cortex-r/ [cit. 10.05.2012]

[3] *Gentoo Linux Documentation: Gentoo Cross Development Guide.* [on-line] www.gentoo.org/proj/en/base/embedded/cross-development.xml [cit. 25.05.2012]

[4] GOVE, Darryl: *Programování aplikací pro vícejádrov procesory.* [kniha] Brno: Computer Press, 2011. 416 s. ISBN 978-80-251-3487-0.

[5] GROEMPING,Ulrike : *Design of Experiments (DoE) and Analysis of Experimental Data.* [on-line] cran.r-project.org/web/views/ExperimentalDesign.html, 2011. [cit. 25.05.2012]

[6] HART D, STULTZ J, TS'O T.: *Real-time Linux in real time.* [serial on the Internet] IBM Systems Journal. 2008, roč. 47, č. 2, s. 207-220. ISSN 00188670.

[7] JELÍNEK, Lukáš: *Jádro systému Linux: Kompletní průvodce programátora.* [kniha] Brno: Computer Press, 2008. 686 s. ISBN 978-80-251-2084-2.

[8] *KernelTrap: Linux The 0.01 Release.* [on-line] http://kerneltrap.org/node/14002 [cit. 13.10.2011]

[9] kolektiv autorů: *BeagleBoard Rev B7: System Reference Manual.* [pdf] BeagleBoard.org, 2008. 162 s.

[10] kolektiv autorů, překlad Lubomír Ptáček: *Linux: Dokumentační projekt.* [kniha] Brno: Computer Press, 2007. 1334 s. ISBN 978-80-251-1525-1.

[11] kolektiv autorů.: *OMAP35x Applications Processor: Technical Reference Manual.* [on-line] Texas Instruments Incorporated, 2012. 3489 s.

[12] KOOLWAL, Kushal: *Investigating latency effects of the Linux real-time Preemption Patches (PREEMPT RT) on AMD's GEODE LX Platform.* [serial on the Internet] RTLWS11, 2009. 28 s.

[13] LAMIE W., CARBONE J.:*Measuring real-time performance of an rtos.* [on-line] Express Logic Inc, 2007. 20 s.

[14] *Linux Documentation: rt-mutex* [on-line] http://www.kernel.org/doc/Documentation/rt-mutex.txt [cit. 08.05.2012]

[15] *Linux man page: pthreads(7) - POSIX threads.* [on-line] linux.die.net/man/7/pthreads [cit. 27.05.2012]

[16] ROSTEDT, Steven, HART, Darren V.: *Internals of the RT Patch.* [on-line] Proceedings of the Linux Symposium, 2007. s. 161-167.

[17] STALLINGS, William.: *Operating Systems: Internals and Design Principles.* [kniha] Upper Saddle River, NJ : Pearson/Prentice Hall, 2004. 832 s. ISBN 8120327969.

[18] STEWART, David B.: *Measuring Execution Time and Real-Time Performance.* [pdf] Embedded Systems Conference Boston, 2006. 15 s.

[19] TIM, Jones M.: *Anatomy of the Linux kernel: History and architectural decomposition.* [serial on the Internet] IBM Article, 2007.

[20] TIM, Jones M.: *Anatomy of real-time Linux architectures: From soft to hard real-time.* [serial on the Internet] IBM Developer Works, 2008. 11 s.

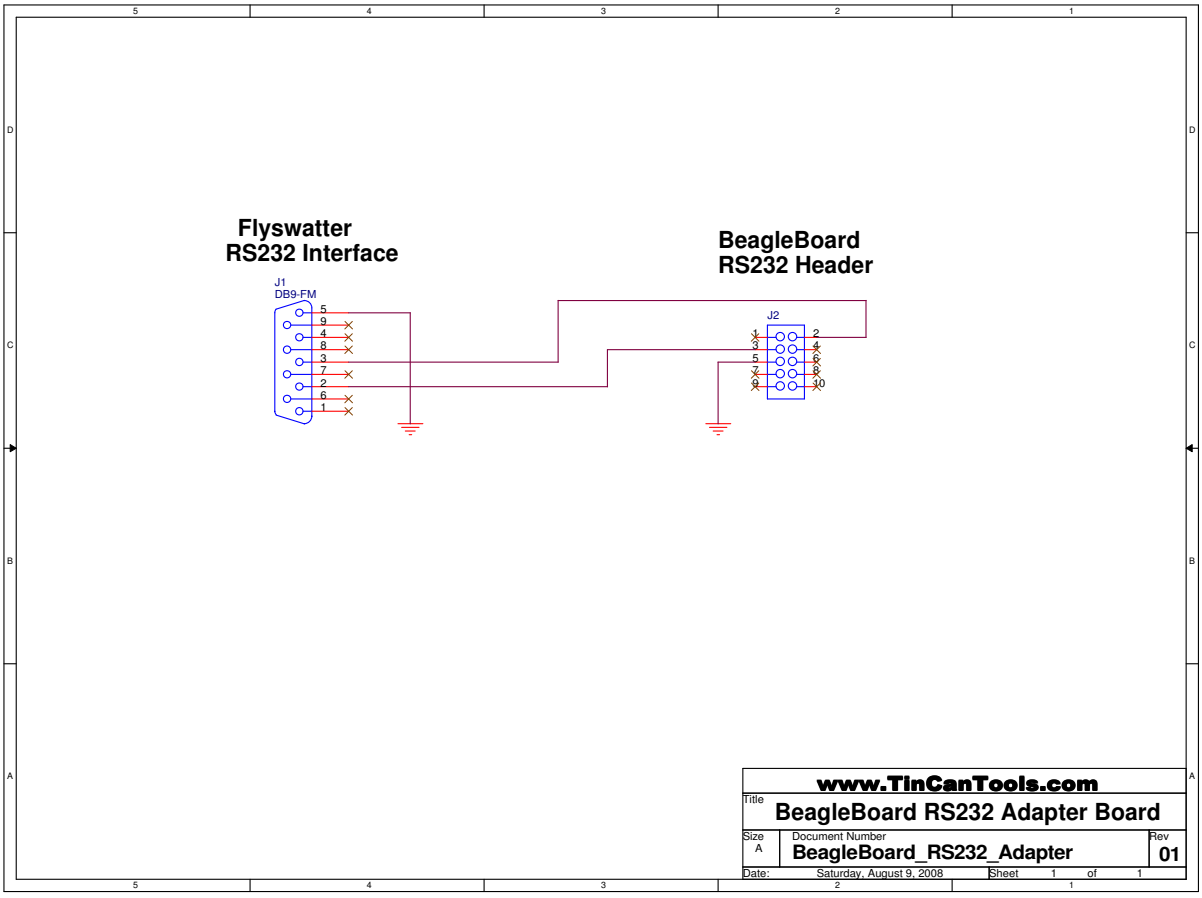
Seznam použitých zkratek a symbolů

A/D	Analog/Digital převod z analogového signálu na digitální signál
ARM	Advanced RISC Machine / Acorn RISC Machine název architektury procesorů
API	Application Programming Interface aplikační rozhraní
CDT	C/C++ Development Tool modul do prostředí Eclipse pro vývoj C/C++ aplikací
CPU	Central Processing Unit nutná součást počítače, která vykonává strojový kód
CCNT	Cycle Counter Register registr vykonaných cyklů
DSP	Digital Signal Procesor digitální signálový procesor, procesor optimalizovaný na zpracování digitálních signálů
EXT3	Third Extended Filesystem žurnálovací systém souborů vytvořen pro OS Linux
FAT32	File Allocation Table souborový systém s 32bitovými adresami clusterů. Zaveden a využíván firmou Microsoft
GPIO	General Purpose Input/Output univerzální vstupně výstupní pin
IEEE	Institute of Electrical and Electronics Engineers institut pro elektrotechnické a elektronické inženýrství
IRQ	Interrupt Reques požadavek na přerušení
ISR	Interrupt Service Routine procedura pro obsluhu přerušení
I/O	Input/Output vstupně-výstupní zařízení
GNU	GNU's Not Unix GNU Není Unix (zkratka je rekurzivní)

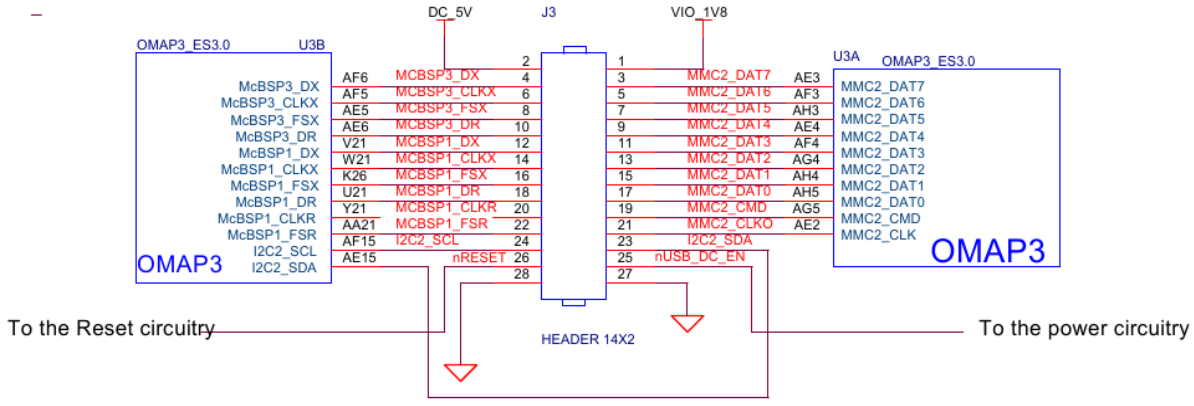
GPL	GNU General Public License všeobecná veřejná licence GNU
MIPS	Milion Instruction Per Second milion instrukci za sekundu
OS	Operating System operační systém
patch	patch záplata, označení pro soupis změn mezi soubory
POP	Package on Package připojení dvou součástek přes svá pouzdra
POSIX	Portable Operating System Interface standart pro zajištění přenositelnosti aplikací
RT	Realtime realtime
RTS	RealTime System realtime systém
RTOS	RealTime Operating System realtime operační systém
SoC	System on Chip integrovaný obvod spojující procesor a většinu potřebných subsystému pro dané použití
UART	Universal Asynchronous Receiver/Transmitter asynchronní sériové rozhraní
UNIX-like	operační systém vycházející z filozofie operačního systému UNIX
USB	Universal Serial Bus univerzální sériová sběrnice, sloužící k jednotnému připojení periferii k počítači
USB OTG	USB On The Go specifikace USB zařízení pro možnost přímého přenosu dat, bod-bod
TI	Texas Instruments Texas Instruments, firma zabývající se výrobou polovodičových součástek
VoiP	Voice over Internet Protocol přenos digitalizovaného hlasu pomocí protokolu IP

Seznam příloh

A. Beagleboard



Obrázek A.1: Schéma zapojení RS232 portu, převzato z [9]



Obrázek A.2: Schéma zapojení rozšiřujícího slotu, převzato z [9]

B. Základní kompilace jádra Linuxu

Příloha v krocích popisuje instalaci jádra Linux a jeho úpravou RT patch. Všechny kroky probíhají v již fungujícím prostředí Linux.

1. Všechny verze kernelu můžeme stáhnout z www.kernel.org. A to buď přímo zkomprimované z webové stránky www.kernel.org/pub/linux/kernel, nebo z ftp serveru ftp.kernel.org/pub/linux/kernel. Další možnost poskytuje program pro distribuovaný systém správy verzí git. Na stránkách git.kernel.org si vybereme cestu k správnému projektu. Pro příklad jsme použili stažené archivy. Pomoc při práci s nástrojem git najdeme na jeho domovské stránce.

2. Dalším krokem je stažení odpovídající verze RT patch. Můžeme postupovat obdobným způsobem, například z www.kernel.org. Například pro verzi jádra 3.2.14 může být odpovídající verze RT patchu 3.2.14-rt1. Poslední číslo závisí na verzi RT patchu.

3. Extrahujeme oficiální jádro do předdefinované složky:

   ```
   cd /usr/src
   tar xjf linux-X.Y.Z.tar.bz2
   ln -s linux-X.Y.Z linux
   ```

   Odkaz na jádro vytváříme jen v případě, že ho chceme používat na daném počítači.

4. Preventivně vyčistíme složku jádra a nainstalujeme RT patch.

   ```
   cd /usr/src/linux
   make mrproper
   bzip2 -dc /usr/src/patch-X.Y.Z-rt1.bz2 | patch -p1
   ```

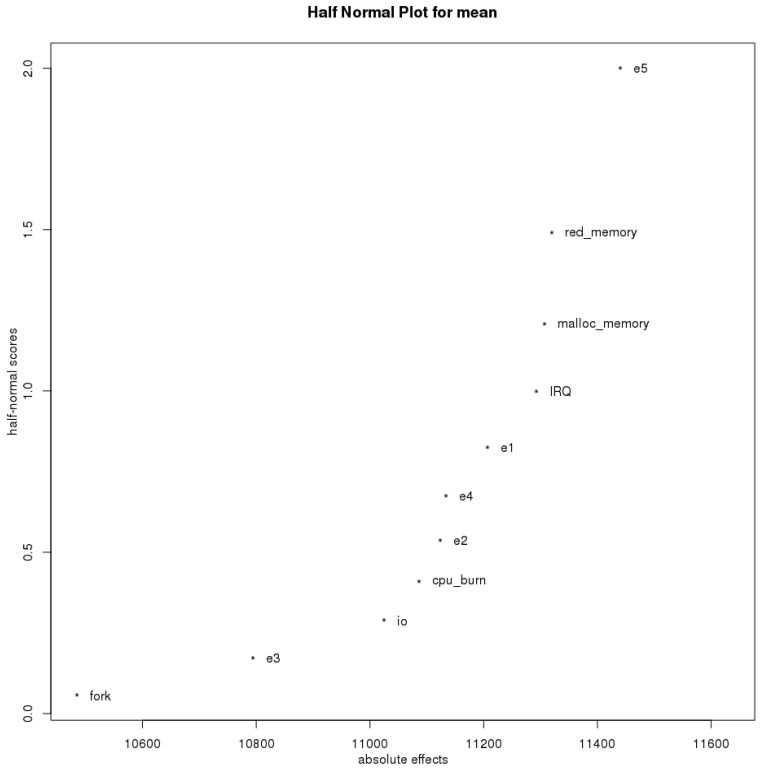
5. Použijeme standardní konfiguraci pro architekturu na které pracujeme. Nebo si můžeme konfiguraci upravit dle vlastních požadavků. Po kompilaci můžeme jádro nainstalovat, skript automaticky zkopíruje jádro do adresáře `/boot`. Také nesmíme zapomenout nainstalovat zkompilované moduly.

   ```
   make defconfig
   make menuconfig
   make all
   make install
   make modules_install
   ```

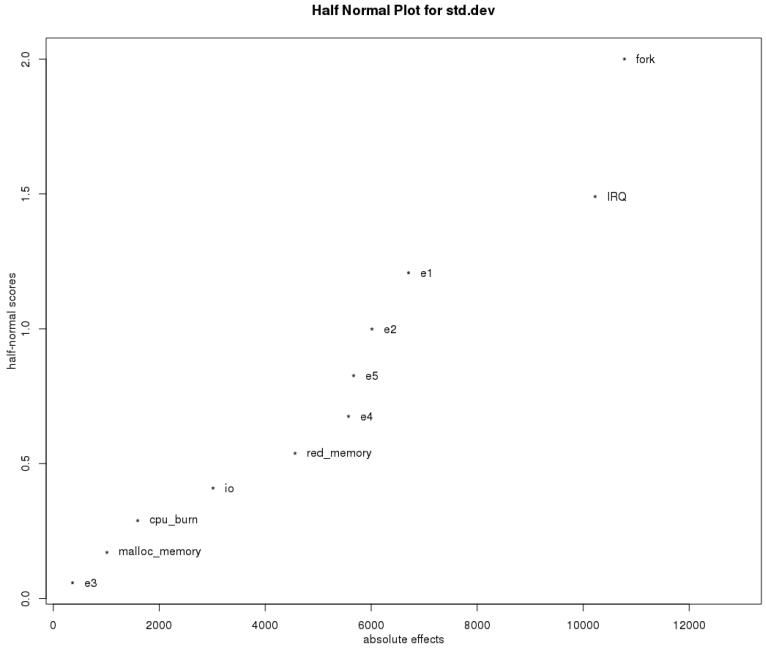
6. Samotné jádro se nachází v adresáři arch/„aktuální architektura“/boot/. Všechny možnosti při sestavování jádra nám vypíše následující příkaz:

   ```
   make help
   ```

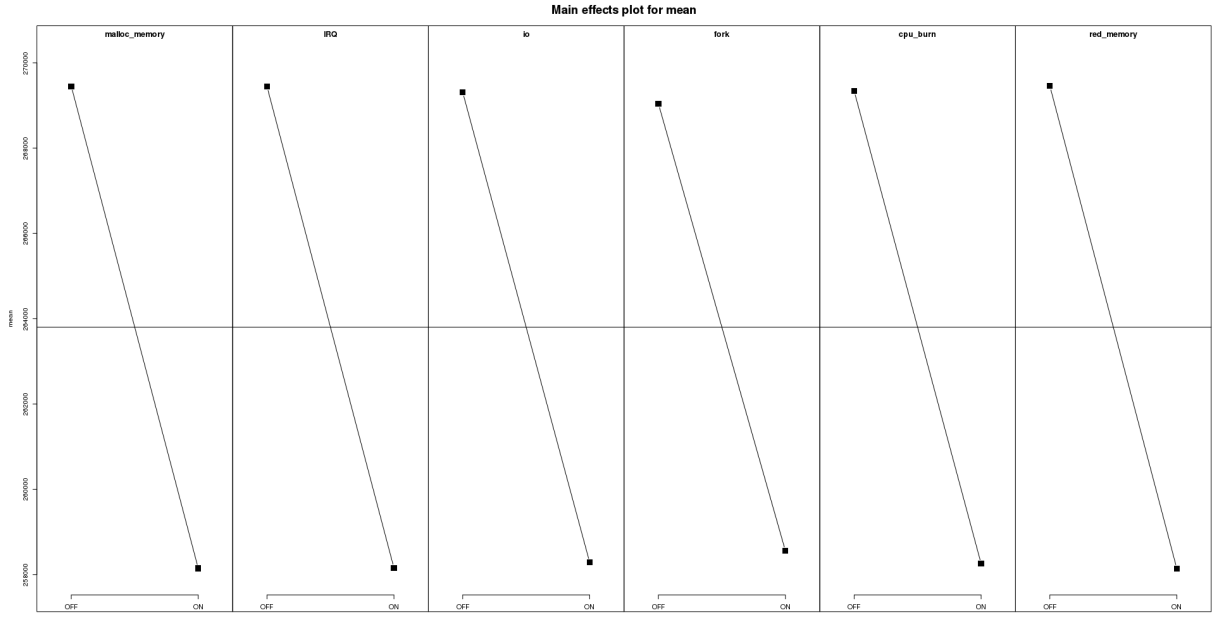
C. Grafy



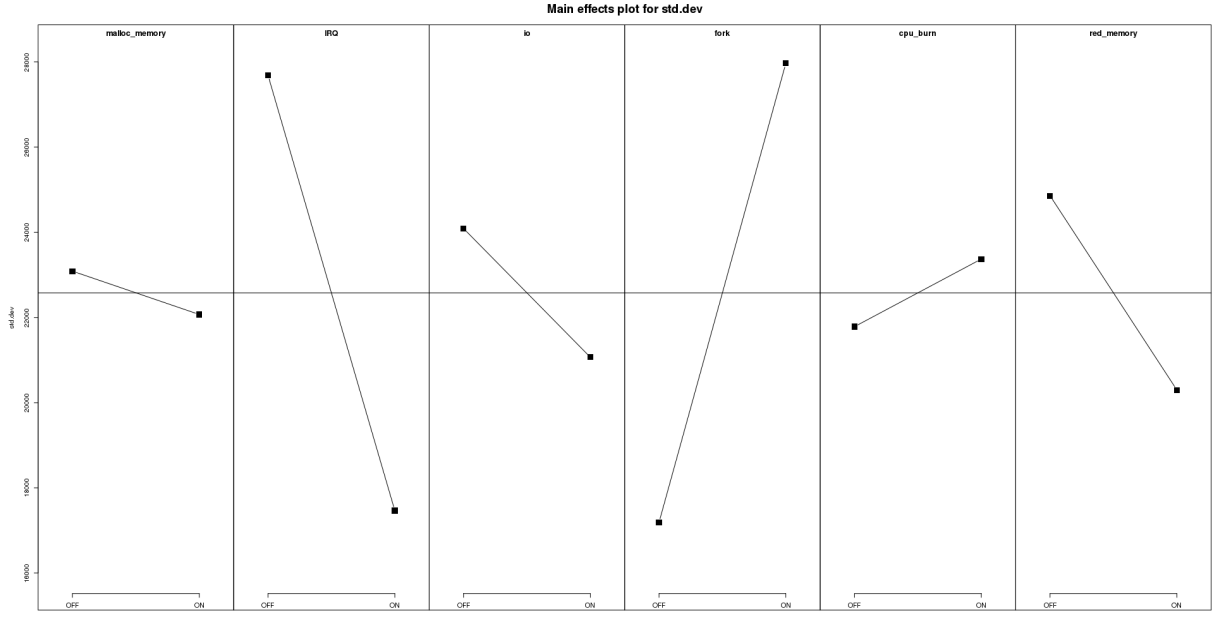
Obrázek C.1: Vliv jednotlivých faktorů na střední hodnotu uspání



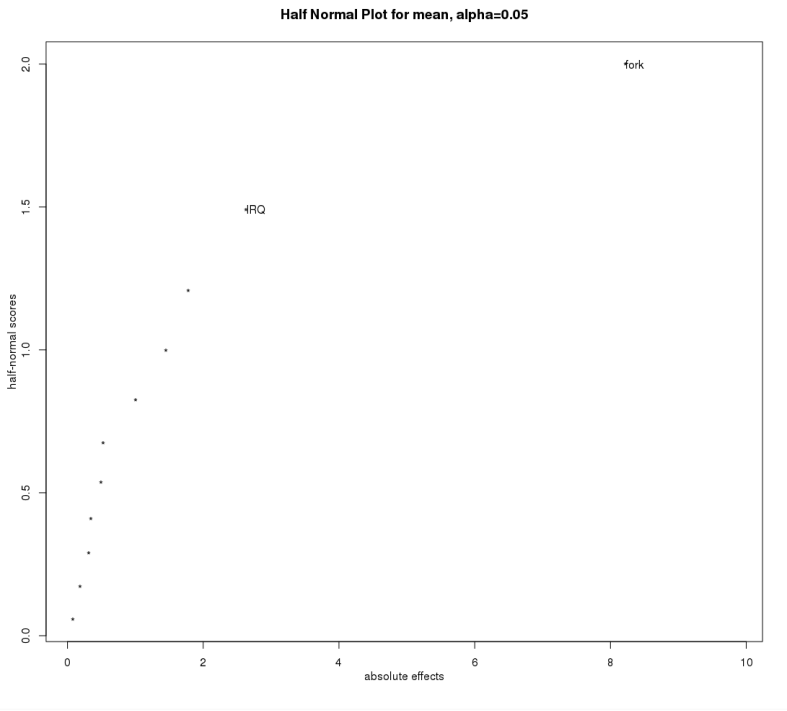
Obrázek C.2: Vliv jednotlivých faktorů na směrodatnou odchylku délky uspání



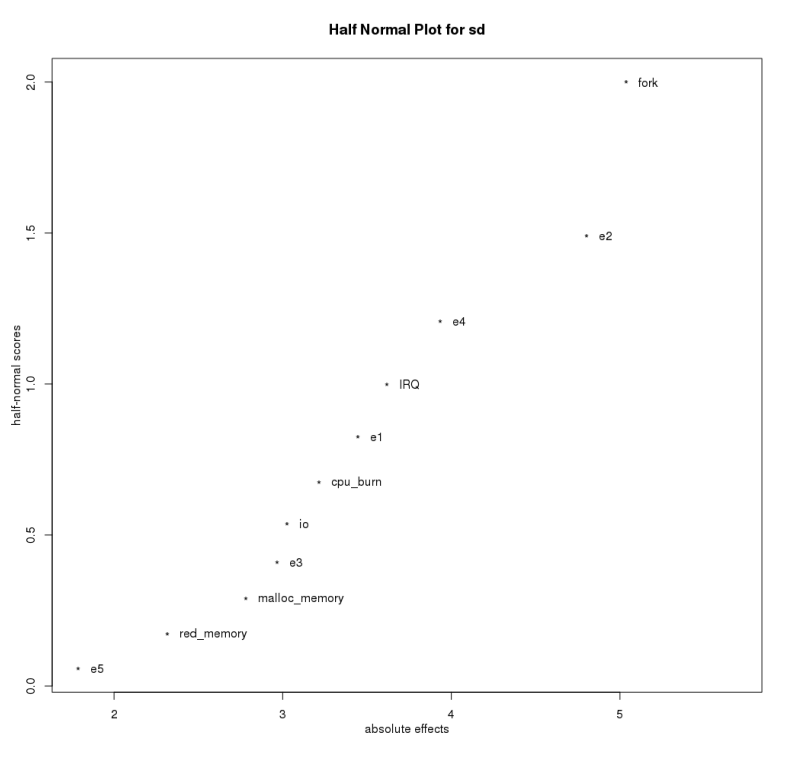
Obrázek C.3: Efekt faktorů na střední hodnotu délky uspání



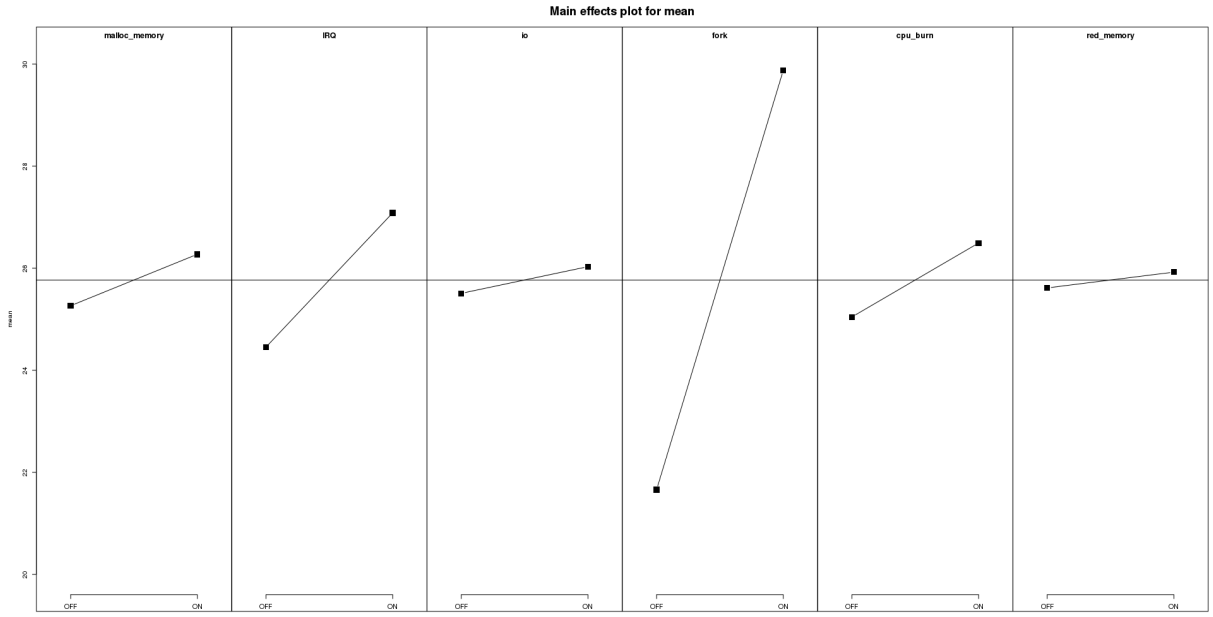
Obrázek C.4: Efekt faktorů na směrodatnou odchylku délky uspání



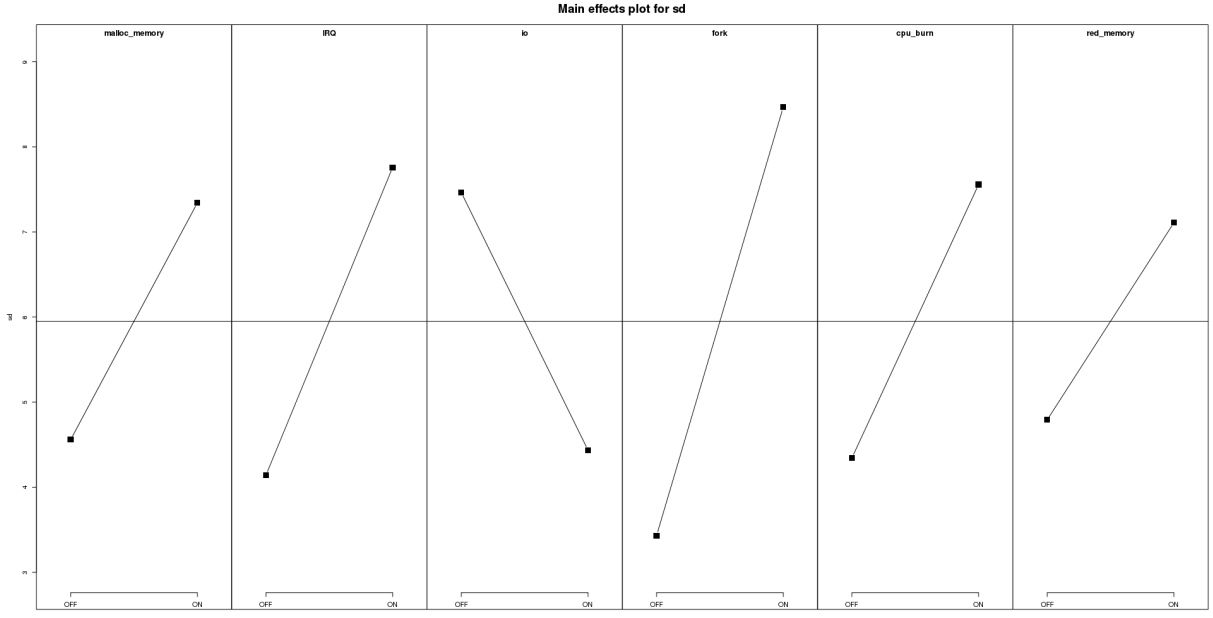
Obrázek C.5: Vliv jednotlivých faktorů na střední hodnotu latence přerušení



Obrázek C.6: Vliv jednotlivých faktorů na směrodatnou odchylku latence přerušení



Obrázek C.7: Efekt faktorů na střední hodnotu latence přerušení



Obrázek C.8: Efekt faktorů na směrodatnou odchylku latence přerušení

D. Obsah přiloženého CD